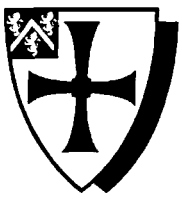
University of Durham



Profiling Large-Scale Lazy Functional Programs.

Stephen Andrew Jarvis

Laboratory for Natural Language Engineering,

Department of Computer Science.

Submitted in partial fulfilment of the

requirements for the degree of

Doctor of Philosophy

Abstract

The LOLITA natural language processing system is an example of one of the ever increasing number of large-scale systems written entirely in a functional programming language. The system consists of over 50,000 lines of Haskell code and is able to perform a number of tasks such as semantic and pragmatic analysis of text, context scanning and query analysis. Such a system is more useful if the results are calculated in real-time, therefore the efficiency of such a system is paramount. For the past three years we have used profiling tools supplied with the Haskell compilers GHC and HBC to analyse and reason about our programming solutions and have achieved good results; however, our experience has shown that the profiling life-cycle is often too long to make a detailed analysis of a large system possible, and the profiling results are often misleading.

A profiling system is developed which allows three types of functionality not previously found in a profiler for lazy functional programs. Firstly, the profiler is able to produce results based on an accurate method of cost inheritance. We have found that this reduces the possibility of the programmer obtaining misleading profiling results. Secondly, the programmer is able to explore the results after the execution of the program. This is done by selecting and deselecting parts of the program using a post-processor. This greatly reduces the analysis time as no further compilation, execution or profiling of the program is needed. Finally, the new profiling system allows the user to examine aspects of the run-time call structure of the program. This is useful in the analysis of the run-time behaviour of the program.

Previous attempts at extending the results produced by a profiler in such a way have failed due to the exceptionally high overheads. Exploration of the overheads produced by the new profiling scheme show that typical overheads in profiling the LOLITA system are: a 10% increase in compilation time; a 7% increase in executable size and a 70% run-time overhead. These overheads mean a considerable saving in time in the detailed analysis of profiling a large, lazy functional program.

Acknowledgements

I would like to express my sincere thanks to the following people:

To my supervisor Rick Morgan for his tireless support and to Roberto Garigliano for his encouragement. To my colleagues at Durham University; James Blowey, Paul Callaghan, Russell Collingham, Chris Cooper, Marco Costantino, Miguel Fernandez, Jon Hazan, Johannes Heitz, Kevin Johnson, Dave Nettleton, Brett Parker, Sanjay Poria, Deborah Robson, Nimish Shah, Sengan Short, Simon Shiu, Mark Smith, Agnieszka Urbanowicz, Yang Wang and Clare Woodward.

To Patrick Sansom and Simon Peyton Jones from Glasgow University.

To my parents, my brother Douglas, and especially Louise.

Declaration

The material contained within this thesis has not previously been submitted for a degree at the University of Durham or any other university. The research reported within this thesis has been conducted by the author unless indicated otherwise.

Contents

List of Figures

Chapter 1

Introduction

When a computer is programmed, for a business system, for an engineering project, for numerical computation or for fun, the programmer is aware of a number of constraints on the solution. The immediate concern is for the program to do the right thing, to produce the correct results in a number of, and preferably all of, the situations in which the program is run. If a program has a unique specification then checking that the results meet the requirements is a matter of course.

The programmer may also be concerned about how the program looks; the neatness of its structure and the fact that colleagues will be able to understand what each part of the program computes. This structure may allow program re-use, or reverse engineering on parts of the code.

The programmer may also be concerned about the efficiency of his solution; that the program calculates the results in sufficient time, that it works for a suitably large input and that the machine on which the program is running does not run out of memory for instance.

The subject of this thesis is the last of these three constraints; the ability of a program to compute the correct result, and the agreeable structure and layout of the program code will be taken for granted.

1.1 Efficient Programs

The analysis of the efficiency of a program can be addressed in two different ways [Runciman and Wakeling, 1992a]. Firstly, this analysis can be done theoretically, that is by reasoning about the way in which the result is computed. This may involve the analysis of the underlying algorithm, recording how many steps would be needed to calculate the results on some abstract interpreter. The programmer can also calculate how an algorithm would perform in the 'worst case'. Analysing the worst case for two different algorithms allows the programmer to choose which algorithm is better, or allows him to discover that whichever he should choose their worst cases are equal.

It is not always true that the worst case is a useful representation of the efficiency of a program. In fact many programs rarely perform solely in their worst cases. It may be more useful, therefore, for the programmer to be able to calculate the 'average' case, based on input for example, which would be more representative of the normal behaviour of the program. Unfortunately the mathematics involved in calculating the average case for the execution of a program are often complicated. Even if the programmer were able to perform this type of analysis, which is not always the case, more often than not the programmer would not be prepared to carry out such complicated analysis.

The second method of analysis is a practical approach. This involves taking measurements using tools while the program is running. These tools provide the programmer with a profile of the program including information such as the time spent in different parts of the program and the memory used during the execution. These tools are known as *profilers*.

Using a profile of a program it is possible to compare two different implementations. One program may run faster than the other, or use less memory space as it computes a result. The programmer may also be directed to parts of the program which appear to be inefficient, contrary to his expectations. These parts of the program can be changed and re-analysed to check for an improved solution.

Because profilers collect their results when the program is running it is easy to check the efficiency of the program with a number of inputs. The programmer may check good, bad and average cases with a degree of ease. No mathematical analysis is needed. Perhaps more important is the ability to analyse the efficiency of large programs which may exclude themselves from a mathematical approach purely because of their size.

1.2 Lazy Functional Programming

Profilers have been built for a number of different programming languages. Profilers for imperative languages such as C have had a good deal of success and are widely used.

Profilers for imperative languages are relatively easy to implement as the order of an expression's evaluation is described as being strict. When a function is called the arguments to the function are evaluated, even if it is not certain that they will be used, and then the function is applied. This defined order of evaluation means that the profiling costs, recorded between the function being called and after the function has terminated, will be the true costs of the execution of that function.

Not all programming languages have this strict order of expression evaluation. Some are lazy. A function's argument will not be evaluated if it is not going to be used by that function and when it is needed, possibly in another function, it will only be evaluated once. Programming languages which exhibit lazy evaluation, such as lazy functional languages, are not as easy to profile. This is partly because the order of evaluation is not as intuitive as it is with a strict language. The profiling costs between a function being called and its termination will not necessarily represent the complete costs of executing that part of the program; the evaluation of some of the functions arguments may be delayed until they are needed, thus moving their evaluation to what appears to be another function.

More recently profilers for lazy functional languages have been developed. Many of the advances have taken place over the past five years. As a consequence of this,

the results of testing such tools on large functional programs are only now becoming apparent. This thesis provides the results of a three-year study of the profiling of a large system written in a lazy functional programming language. In response to these results a number of improvements to the field of profiling are proposed. These improvements are implemented in a new profiling tool which is tested within the domain of the natural language processing system known as LOLITA.

1.3 The LOLITA System

The Laboratory for Natural Language Engineering at the University of Durham has developed the LOLITA system for natural language processing applications. The system consists of 50,000 lines of source code (not including comments; with comments, the system consists of approximately 80,000 lines), equivalent to about 500,000 lines of imperative code [Turner, 1982], divided between 170 modules. In addition there are around 450 data files. Although the system was initially developed by one person, a team of approximately twenty people is currently engaged in developing various aspects of LOLITA. The LOLITA system is written entirely in the lazy functional programming language Haskell.

This thesis contains a study of the profiling of the LOLITA system over a number of years. This study is unique for a number of reasons. Unlike many other large lazy functional systems LOLITA is written in a lazy functional language because the language was considered best suited to the task. The aim was not to experiment with the suitability of lazy functional languages for particular programming tasks, nor is LOLITA a system related to the use of functional languages such as a compiler. The development team of around 20 programmers are specialists in a number of areas such as mathematics, linguistics, economics and computing. They are not all programmers. Some have little or no programming experience when they join the project. For this reason the testing of the current profiling tools found in this thesis is based purely on the effectiveness of the profiling tools for the development of a very large functional program. This work differs from that of Sansom

[Sansom, 1994] which focuses on the profiling of the Haskell compiler GHC and also the work by Runciman and Wakeling [Runciman and Wakeling, 1993] in which the programmers have experience in the implementation of functional programming languages.

1.4 Contributions of the thesis to profiling

The series of profiling experiments performed using the LOLITA system over a period of three years has highlighted a number of problems with the current profiling tools:

- A functional program is usually composed of a number of small functions. This can often cause problems when profiling large programs as it can be difficult to pinpoint exactly where an efficiency problem manifests itself in the code. One of the problems with profilers is that they have presented their results at a low level in the code, reporting a large percentage of the execution costs as a result of supporting or library functions. This is of little help in the identification of an inefficient algorithm, particularly for programmers who may be programming at a much more abstract level in the code. Alternative methods of profiling allow the programmer to identify which parts of the code he is interested in profiling. This gives the programmer more flexibility over where in the code the results are displayed. However, existing schemes have relied on the programmer recompiling and profiling the program each time he wishes to shift his attention from one part of the code to another. This may cause significant problems.

- Functional programs use a large number of shared functions. It is often useful to identify which parent functions make most use of these shared functions, allowing the path of expensive function calls in a program to be identified. To date no accurate scheme exists; statistical averaging has been used in some implementations of profilers, alternative implementations have no form of cost inheritance at all.

- Profiling takes a long time, particularly if the programmer is required to identify the parts of the code which he is interested in profiling. Tracking down an efficiency problem in the code can require a number of recompilations and executions of the program. The result is a task which, with a large program, can conceivably take days to complete.

In response to these problems a number of improvements are proposed and implemented:

- An accurate method of cost inheritance is introduced which allows the paths of expensive function calls to be identified. All of the costs in the program can be identified, in particular shared functions have their costs accurately divided between the parent functions. This scheme is implemented as part of the standard profile of a program.
- Existing profiling semantics are extended to model this method of profiling. This allows the description of the mapping of execution costs to individual parts of the program to be mathematically defined. This is useful for more formal reasoning about a program's execution costs.
- A new profiling system is implemented on the Glasgow Haskell Compiler; modifications are also made to the compiler optimisations so that the efficiency of the compiled code is not affected by the introduction of the new profiler.
- A post-processor is designed to allow programmers to manipulate the new profiling results after the execution of the program. This means that the program does not have to be recompiled in order to view the profiling costs at a different part in the program.
- The post-processor allows the results of the existing profilers to be produced. It also allows the results to be displayed as inherited costs. This prevents any ambiguity in the presentation of results of shared functions and provides a useful device in allowing the programmer to find the cause of efficiency problems in his code.

1.5 Criteria for Success

The success of this work will be evaluated in terms of providing evidence for or against the following statements:

1. The new method of profiling presented in this thesis provides an opportunity for a reduction in the time needed to profile a large lazy functional programmed system, when the programmer selects and deselects parts of the code for profiling.

2. The new method of profiling extends the profiling results presented by previous profilers in so much as the accurate inheritance of shared program costs can be achieved.

3. The new method of profiling provides these new facilities without imposing an unacceptable overhead on the compilation or execution of the program such that the new method of profiling would offer no benefit to a functional programmer.

1.6 Thesis Structure

The thesis is divided into nine chapters.

Chapter 2 considers in detail the efficiency analysis of programs; introduced in section 1.1. It considers in more detail the questions involved in the theoretical and practical analysis of efficiency in programs. The criteria for a profiling tool are discussed and the differences between profilers for imperative, logical and functional languages are analysed. The different types of profiling tool are categorised and any theoretical considerations in their design are introduced. Some of the current profiling tools for imperative languages are discussed; these provide a context for looking at the more recent profilers for functional languages.

Chapter 3 discusses the programming of large-scale functional systems and in particular the experience gained in the development of the LOLITA system; intro-

duced in section 1.3. The LOLITA system is one of the few real-world functional programs which is not implicitly related to the development of functional programming languages. For this reason there are many aspects of the programming of the system which make it unique. The programmers of the system are taught to program at an abstract level in the code, which facilitates the production of specialised code for the natural language processing capabilities of the system. This also provides a unique insight into the needs of applications programmers in a large, lazy functional program. The chapter ends with a discussion of the programming tools which are needed for system development.

Chapter 4 provides a number of case studies of the profiling of the LOLITA system. These case studies are chosen from a two-year period of data collection. They highlight the problems which the LOLITA programmers have experienced with the current profiling tools supplied with the Glasgow Haskell Compiler and the Chalmers Haskell Compiler. Proposed improvements are made to the existing profiling tools based on the information collected from these case studies.

Chapter 5 discusses the implementation of the existing cost-centre profiler on the Glasgow Haskell Compiler. This profiler is used as a basis for the changes which are proposed in the previous chapter. In order to understand the details of the proposed changes, the cost-centre profiler is described in part. These details include the cost-attribution semantics and the core language optimisations performed by the compiler simplifier.

Chapter 6 contains the description of a new profiler, the cost-centre-stack profiler, built on top of the existing cost-centre profiler. This allows the proposed improvements to be implemented in a profiler on the Glasgow Haskell Compiler. The cost-attribution semantics seen in the previous chapter are extended. Further semantics are introduced to model the behaviour of cost inheritance. In this chapter the efficient implementation of this new profiler is also discussed. This ensures that the profiler has acceptable overheads despite collecting considerably more profiling information. The implementation with the Glasgow Haskell Compiler is also discussed. Compiler transformations are modified so that the new profiler preserves

the efficient code produced by the compiler optimiser. Finally a post-processor is described which allows the new results to be combined in an effective programming environment. This is a new way of presenting profiling results and allows the programmer to view the effects of accurate cost inheritance.

Chapter 7 contains the results of the new profiler for small and large programs. Attention is paid to the new results which the cost-centre-stack profiler produces. It is shown how these can be useful in helping the programmer to identify the path of function calls to computationally expensive parts of the program. The chapter also investigates the overheads which this new method of profiling imposes and whether these overheads make this method of profiling feasible.

Chapter 8 considers further research. The method of profiling presented in this thesis is applicable to two other important areas in the analysis of lazy functional programs; debugging and tracing. It is proposed that the new theory introduced to implement the cost-centre-stack profiler will also provide a unique solution to these further fields. In addition it is discussed how the profiling theory is also being applied in the development of further profilers.

In Chapter 9 the conclusions of the thesis are presented.

Chapter 2

Efficiency Analysis of Programs

The future paradigm of programming may well be clearly established; to create software a specification is first written according to the user's directives, and then the specification is refined to an implementation which is clear and efficient. In particular, this paradigm is a prime motivation behind the study of functional programming. Much has been written about the process of transforming one functional programming language into another [Ajisaka, 1987][Burstall and Darlington, 1977][Darlington and Burstall, 1976][Loveman, 1977]. However, a key part of the process, assessing the efficiency of functional programs, has, until recently, remained largely untouched.

Computing takes time. Some problems take a very long time; others can be done quickly. Some problems *seem* to take a long time, and then a faster way to do them is discovered (a faster algorithm). The study of the amount of computational effort which is needed to perform a calculation is the study of computational *complexity*.

It is particularly important in the context of real-world applications that a programmer is able to record the demand on the necessary resources needed for the execution of a program. In a real-world environment there may be limits on the resources available and often these resources will be shared between different people or programs. By studying and analysing the complexity of a program, it is possible to understand the demands which are placed on the available resources.

Improvements can then be made to the program and efficient and effective solutions are produced.

Measuring the complexity of a program also highlights those parts of the program which perform differently from the programmer's expectation. Performance bugs can be identified in the execution of a program. An error in the algorithm design or the refinement of a specification can be identified at the program level. This analysis enables the relationship between the program's code and its resources to be understood by the programmer. There are therefore a number of benefits from doing this type of analysis.

If the complexity of a program is to be classified, a scale on which complexity constraints can be measured must first be established [Wilf, 1986]. A program is considered complex if it is based upon a complex algorithm. In turn, an algorithm is considered complex if its application requires the execution of a complex computation. A computation is a calculation performed by the machine, usually in the process of applying a part of the algorithm to a particular setting in order to get a result.

To fulfill the goal of measuring the complexity of programs, therefore, it must be possible to measure the complexity of individual computations; from this the complexity of algorithms can be determined and finally the complexity of programs.

What is complexity? Although complexity can be intuitively understood, a precise definition is needed if a scientific investigation is to be carried out. Intuitively, a computation is *complex* if it is difficult to do. But how can difficulty be measured? A common approach, and the one to be followed here, is to measure the complexity of a computation by measuring the resources required to execute it — it being assumed that a difficult computation will require more resources to carry it out.

One resource often used in this context is time. One computation is considered to be more complex than another if the execution of the former takes more time than the latter. The amount of time required to perform a computation is called the *'time complexity'*.

Another resource often used to measure complexity is the amount of storage space required. This is based on the assumption that the more difficult a computation is, the more space will be required for its execution. The amount of storage required by a computation is referred to as the *'space complexity'*.

2.1 Measuring Complexity

The problem of reasoning about the time and space complexities of programs can be addressed in two different ways; the *theoretical* and the *practical* approaches [Runciman and Wakeling, 1992a].

2.1.1 The theoretical approach

The theoretical approach is described as developing a framework in which a programmer may reason about a program's intensional properties (how a result is computed) using algebraic methods similar to those used to reason about its extensional properties (the result of the computation).

Theoretically, the 'time cost' of a program can be represented as the number of steps an abstract interpreter needs to compute its result. One way of computing this time cost would be to simulate the abstract interpreter and count the number of steps it would need; alternatively, the time cost of a program may be expressed in terms of the time cost of its parts, in a more 'compositional' manner. This latter method is essentially more difficult in lazy functional programming as the reduction steps must also consider how much of the result is required.

The theoretical approach is concerned with providing a method of converting a functional program into a series of equations, a necessary precursor to automatic analysis. These time equations can be solved by traditional methods (complexity measured in big-O notation for instance), yielding either an exact solution, an upper bound or an approximate solution.

There has been a collection of work in this area, [Wadler, 1988] [Bjerner and Holmstrom, 1989] and [Sands, 1991], which is reported as having had a moderate amount of success [Runciman and Wakeling, 1992a]. One of the deficiencies of this technique is that reasoning about large programs can be a lengthy and difficult task. Automation of this theoretical method of time analysis would solve this problem but how successfully it might be accomplished has yet to be explored.

Theoretical analysis is also limited by the fact that in most cases the calculations are worst case analyses; the average case is in many situations too difficult to calculate. For many programs the analysis may therefore be misleading, as systems spend most of their time in a 'better than worst' state. Even when the worst case is the normal one, the complexity analysis concerns the input as it grows towards infinity. Most systems will have a restriction on their input, though this restriction is not considered in the analysis of the order of complexity.

In many cases the theoretical approach will be useful in the analysis of algorithms, the results of which can be used to determine whether one algorithm is preferable to another. Programs however, may require a more practical approach to complexity analysis, so that constraints such as average case analysis and finite input can be considered.

2.1.2 The practical approach

The practical approach, which is the subject of this thesis, considers tools which gather statistical information about the use of resources while the program is being executed. The tools provide us with a *profile* of the computation and inform us of information such as how often each statement is executed and the amount of time spent in each part of the program code.

These statistics allow the programmer to see which parts of the program are executed most often and for how long their execution lasts. In addition they permit the identification of those parts of the program which are executed infrequently or not at all! The run-time behaviour of the program can be shown and consequently

the programmed solution can be examined in terms of its run-time complexity.

The program can be executed with a sample of data which represents an average input. Many of the efficiency improvements may be made according to the results of such a profile, as a large proportion of the system time will be spent processing such cases. Of course this may mean that the program takes a large amount of time to process data in a 'worst case', but providing such a case is rare, this may not be of importance anyway.

Tools which provide such a profile allow the non-theoretician to analyse the behaviour of his programs. These tools also allow the experienced programmer to speed up the efficiency analysis of his code. A majority of programmers will in preference choose 'automatic' over 'hand/paper' analysis of their programs. This observation is particularly true within the domain of computing. Programs are often so shrouded in the beauty of their autonomous behaviour that the designer is happy for them to remain un-quantified, until of course something goes wrong! Profiling tools are now examined in more detail.

2.2 Tools for Practical Complexity Analysis

When programs are written, the primary concern is the correctness of the code. Any debugging which is performed is mainly to correct logical errors. The result is a program which meets the initial correctness requirements, or specification. The measurement of the complexity of the program, to determine its efficiency, is often considered secondary to this correctness, and it is not until the programmer is happy with the correctness of the code that he begins to consider the complexity of his solution.

The first attempt at the efficiency analysis of a program might often consist of recording the execution time and perhaps the amount of memory needed to run the program. These two metrics give a profile of the program; a tool which measures these execution costs (and others) is therefore called a *profiler*.

Profiling tools have been developed which provide more comprehensive information. There are various methods of gathering profiling information, these are described in section 1.4; first some of the more general problems involved in profiling are considered [Sansom, 1994][Platter and Nievergelt, 1981].

In developing profiling tools there are some general principles which should be noted:

- What is measured — There are a number of different resources which can be measured. Memory space can be divided into the memory used for the program stack, which is used for procedure calls and argument values, and that used for the program heap, which is used as working memory for the evaluation of expressions. CPU time can also be measured. This may be the time used by the program on a single processor, or the total time used if a number of parallel processors are utilised for the program evaluation. Other operating system measures can also be recorded such as: network communication; number of file handles; systems calls; and garbage collection. This is discussed in more detail in section 2.2.1.

- How this information is recorded — As well as knowing what to record there must also be an effective way of gathering this information. The general problem of monitoring a program's execution (that is, observing the empirical behaviour of programs) and the particular problem of observing program performance is addressed in section 2.2.2.

- Presenting the profiling results to the programmer — Recording information using a profiling tool is only going to be successful if the results are presented to the user in some interpretable form. This will require displaying the results in some way that is meaningful to the programmer; section 2.2.3.

- What program data should be used — With the introduction of practical complexity analysis comes the choice of what input data should be used when the program is tested. Some data should be chosen to represent the 'average case' of input. Data can also be selected for 'bad case' analysis or even 'worst

case' analysis, although the distinction between this and the *infinite* worst case of theoretical complexity analysis should be noted; section 2.2.4.

- Relating resource usage to the source code — While sampling the resources used during the execution of a program, it is necessary to relate these costs to units of the program source code. This relationship between the costs and units of the source code may follow a number of techniques; section 2.2.5.

- Preserving the original behaviour of the program — The profiling tool should maintain the program's original behaviour. Transformations and optimisations which take place during the compilation and execution of the program should be preserved whilst profiling. The profiler should not, as far as possible, increase the resources needed by the program for its execution. Or, at least, any extra resources which were needed in profiling the program should not be included in the profiling results; section 2.2.6.

- Programming paradigm — The method of collecting the costs of a program's execution and the way in which these results are presented will depend on the programming language in which the source code is written. For some languages which are designed to be more abstract, such as functional languages for instance, this problem becomes even more pronounced. Delayed evaluation schemes introduce problems of where the costs of the evaluation of an expression should be attributed; section 2.3.

The remainder of this chapter is concerned with the discussion of these principles and of some of the profiling tools currently available.

2.2.1 What is measured

A profiler is only worthwhile if it produces relevant information for the user to interpret. It is therefore necessary to consider carefully exactly what metrics should be recorded.

The first and simplest results which can be recorded are the *number of calls to portions of the code*; although the number of procedure calls will not necessarily correspond to the amount of resources used, so for instance 'more calls' may not necessarily mean that 'more resources' are used. However, a lower limit can be established if we consider that each call to a function requires a fixed amount of space and a fixed amount of time in which to be executed. It may also be useful to the programmer to know that some parts of the code are called considerably more than others.

This method is undeniably crude, as clearly some parts of the code will take longer to execute than others. Therefore, the *amount of execution time* at each part of the program may also be measured. The frequency counts can then be augmented with the percentage of total execution time that this part of the code took to execute.

Execution time is established as the amount of CPU time dedicated to the evaluation of a piece of code. This time should be free from any time-sharing properties that the underlying operating system may exhibit. The time may come from a single processor or may be the collection of results from multiple processors.

The third measurement considered is the measurement of *memory usage*. This allows the programmer to 'see' the allocation of memory to individual parts of the program, a process which is typically hidden in the language implementation, in particular in those languages which use dynamic storage management. Memory costs may for instance be recorded in terms of individual functions, modules, or language constructs.

The measurement of time and space usage can be extended. During the collection of cost data it is also possible to record the point in time at which this measurement is taken. Thus the results are a collection of pairs containing the program cost and the point at execution time in which this cost was recorded. This is useful because, as well as recording the fact that a function uses a large amount of memory, it may also be useful to the programmer to know *when* the function used this memory.

It may also be useful to include in the profiling results some of the operating system overheads such as file manipulation and transfer, communication costs, garbage collection and the division of tasks to processors. Language-specific issues such as expression reductions and the number of times different parts of the internal representation of the code are visited may also be useful metrics to keep. This enables the programmer to gain an overall picture of the execution costs.

One or a number of these measures can be adopted for a successful profiling tool. Often the application which is being profiled will have specific efficiency needs; a fast execution time, a low memory usage, or both. The appropriate measure can therefore be selected. A combination of these features is often presented.

2.2.2 Recording profiling information

Recording the profiling data requires some method of collecting information about the resources used during expression evaluation, then a way of relating this information to the unit of code which caused the evaluation to take place.

This information can be gathered using two approaches.

One method of profiling, termed *Execution sampling*, is based on sampling operating system signals. The execution of the program is interrupted at regular intervals (every 20ms for example). At this interrupt, the interrupt handler can increment the resource counters which are associated with the piece of code that the program is currently executing. So for example, time profiling may be implemented by the interrupt handler incrementing a time counter associated with that code unit which is being executed. Memory profiling can be implemented by scanning the memory cells to detect which are currently being used by that part of the code.

At the end of the program's execution the sample for each piece of code will be divided by the total number of samples, thus producing a percentage of the total time or memory spent executing that part of the program. This is an effective method of dividing execution costs between parts of the program, though it does have drawbacks. The execution time must be long enough to have a reasonable

number of samples; if not, the results will be inaccurate. Also, as the samples are not controlled by the code being executed, it is possible that two runs of the same program may produce slightly different results.

A second method, termed *Procedure timing*, uses samples which are instructed by the program code itself. Statements can be inserted into the code which indicate a read of the system clock or a scan of the memory currently in use. These costs can then be compared at two points to give the total amount of time and the total memory used in a certain part of the program. This method enables the resources used in each procedure, either including or excluding any sub-procedures, to be determined. Unfortunately, the cost of accessing the system clock is often prohibitively expensive and the accuracy of the profile is dependent on the clock resolution. However, it is easy to separate the profiling costs from the program costs, as the sampling is linked directly to the execution of the code.

2.2.3 Presentation of the results

After the profiling data has been recorded there is the separate task of presenting the data to the user. Numerical information such as percentage values or the number of calls to functions can be presented in a number of ways:

- The information can be displayed in a table, in a textual form. This will contain the names of functions (or units of code) and the corresponding costs which they incurred.
- Alternatively the source code itself can be annotated with the results of the profile. The results will still be in a textual format.
- Results can also be displayed as a graph. This method can capture more information than the textual results as it can also show *when* in the program's execution the costs were recorded.
- Finally the results can be displayed interactively. This may involve some method of program representation, such as a call-graph, and the display of

costs at selected points in this graph. Using this approach the programmer can trace costs through the program by selecting and de-selecting parts of the code as required.

Displaying the results in a table is a useful way of summarising large amounts of information; it presents the data in a readable form, particularly if the program is large. However, it does require the programmer to identify exactly where in the program the code unit is defined.

Annotating the code with the profiling results saves the programmer from having to map between the results and the source code, often a difficult task. This method does however become more complicated to implement and interpret the larger the program becomes. Identifying the parts of the program with high costs may require trawling through large amounts of code trying to filter out the expensive parts from the rest.

Presenting profiling information in a textual form can be useful for both time and memory profiles. The results can be stored in tables and can be ordered or grouped by logical structures, such as module names for example. A programmer may require a number of program profiles before he is satisfied with his interpretation of the program results.

A second dimension is added by considering 'when' in the program's execution these time or memory costs are collected. Such results can be presented graphically as *serial* profiles, showing cost over time, or may be displayed in real time as the program is executing. These give the user timed information which may make correlating programming problems to points in the execution, and ultimately to sections of the source code, an easier task.

An interactive presentation of results aids the programmer by allowing him to explore the profiling results after the execution of the program. The user may do this by following expensive results through a program call-graph. This has the advantage of reducing the number of program profiles needed before a satisfactory interpretation of the results is achieved.

Interactive analysis has the benefit of displaying different parts of the results as and when the user needs them; this also highlights the relationship between program units and costs. In addition it means that a single set of results can be collected and viewed in many different ways; the final result of the profiler is not static. This method is explored in more detail in this thesis.

2.2.4 The choice of run-time data

Profiling tools are sensitive to the data with which the program is run. Often the programmer will be interested in a general case, how the program performs when given an input of average data. In such a case there may be little attention paid to the program data supplied; however, there are a number of interesting points which should be noted.

- There may be a point in the execution of the program when the complexity of the behaviour changes. A search of a list of size n may be linear if n is below a certain threshold; above this threshold the complexity of this search may become logarithmic.

- The program which is being executed may have a wide range of functions. If this is the case then the profiler can be used in more than one way. Either test data can be given to the system which will force a wide coverage of the functions available, or the system can be tested more specifically for one, or a collection of functions. The important thing to note is which of these tests is being applied.

- The programmer may also be interested in more specific scenarios such as 'how the program performs with the input n'. These extraneous cases may not be representative of the complexity of the whole program and the programmer may need to consider some universal cases for comparison.

- Some systems require a certain amount of time to set-up. The resources required for this set-up may not be important to the programmer, as such

an operation is only a one-off cost. If this is the case, it will be necessary to profile the system over a period of time long enough for the test results not to be skewed by the set-up operation.

These cases will affect the results. It may be necessary therefore to understand exactly what the program is being asked to do before analysing the profiling results.

2.2.5 Relating resource usage to the source code

A profiler will only be useful if it accurately maps the resource usage back to units in the source code. This is the primary consideration; it does not matter how the results are displayed or interpreted by the user if they are incorrect.

There are two issues to be considered. Firstly it is necessary to understand *what* program *units* the costs are to be mapped to. These units may simply be function or procedure names. They may also be data types or values in the program, each of which can also be identified by its name. The cost of resources needed for a part of a program will be associated with a counter of the same name.

Secondly it is necessary to identify *which costs* will be attributed to these program units. The costs at one unit in the program may or may not include the costs of all its sub-units. This distinction is defined by the terms *flat* profile, which refers to the cost of the single function not inclusive of any sub-functions; and an *inherited* profile which may include the costs of sub-functions in the cost of the single function.

Shared sub-functions complicate the task of inheritance, as it is necessary to understand what percentages of the shared sub-functions costs should go to its parent functions. This percentage figure may be allocated accurately, *accurate* inheritance, or by a statistical approach, *statistical* inheritance. This will be discussed in more detail in section 2.5.

Another problem arises due to the fact that it may not always be obvious which units give rise to which costs. Delayed evaluation may mean that arguments

are passed through the program and are only evaluated if and when they are needed. The cost of evaluating the arguments may be associated with the part of the program in which the arguments are defined, or that part of the program which cause their evaluation. These two program units may be very different, and thus the results which the profiler produces will be different.

2.2.6 Maintaining the program's original behaviour

An essential property of any profiler is that the profiling information gathered must be faithful to normal execution [Sansom and Peyton Jones, 1994]. In particular:

- The evaluation order of expressions must not be modified;
- Actual events during execution should be considered, an instrumental interpreter would not prove satisfactory;
- The profiling scheme must be reasonably cheap, additional costs created by the profiler should not prevent larger examples from being profiled in an acceptable time;
- The additional cost of profiling should not distort the information gathered about the execution.

It would not be acceptable for the program's behaviour to change between profiled and un-profiled versions as the monitored version would then be a misrepresentation of the program's original behaviour. A program's behaviour can be influenced in a number of ways. Firstly, the effect of having a profiler running in parallel with a program should not affect the way in which the program's results are calculated; the profiler should, as far as possible, be an unobtrusive observer. Secondly, when the program is run it must have exactly the same run-time flags as the non-profiled version of the program. This must include the preservation of the same garbage collection scheme and the same compiler optimisations, the latter making relating the costs back to the source code a difficult task. The solution to this problem is to

build profilers around program compilers so that they can recognise or compensate for any changes in code during the optimisation stage of compilation. Finally, the profiling overheads themselves should not be included in the final results. Previous profilers have had overheads of between thirty and one hundred percent on the execution time of a program. These overheads are far less important if the results of the program's execution and the results of the profiler's execution are kept separate.

2.3 Different programming paradigms

There are a number of features common to different profilers, both those specifically written for imperative languages and those written for declarative languages. Many of the differences between profilers for different language paradigms are created by the necessity to compensate for any unique language features which the source code exhibits. There will also be a difference in basic program 'units' and the allocation of resource costs to them.

2.3.1 Imperative Languages

Profilers have long been available to programmers of imperative languages. There exist many instances of the use of profilers with languages such as Fortran, Ada, Pascal and C [Bentley, 1988].

Profilers for imperative languages have produced impressive results. For example, Kernigham used a line-count profiler supplied with the AWK interpreter on a 4000 line C program. The results showed that most of the functions had been annotated with a few thousand calls, but that one part of the code, an initialisation routine, had over a million calls. By changing these six lines the program speed was doubled. Knuth described how a line-count profiler was applied to itself. The profile showed that half the runtime was spent in two loops. Changing a few lines of code doubled the speed of the profiler in just a few hours' work.

More specifically Knuth concluded from a collection of examples that "less than four percent of a program generally accounts for more than half of its running time".

Designing profilers (especially those which measure how often code is called) has been shown to be simple for imperative languages. Imperative languages have a defined sequential order of instruction execution and are evaluated strictly rather than on demand. This strictness means that it is valid to insert points in the code and measure the time that it takes to execute the code between the two points. In this way it is possible to work out how much resource has been used 'so far'. By measuring the resource usage before and after a segment of code, it is possible to determine how much resource is used in that part of the program. This simplifies the construction of tools to monitor program behaviour since it is possible to identify code blocks and the order in which the code will be executed.

Once the results from profiling an imperative program have been gathered, relating the results back to the original source code is a straightforward task. A direct relationship exists between the object code and the imperative source code, as the very nature of the imperative code means that it is related to the underlying execution engine.

2.3.2 Logical Languages

Traditional computer architecture is not particularly suited to a logical style of program execution, that is, satisfying a list of goals. Therefore logical programmers may quickly find that time and space usage may prevent the development of practical applications. This is perhaps contrary to the actual system development time, as logical programs tend to gravitate towards a symbolic non-numeric process, based on data objects and relations between them. This, it is argued, speeds up the time in which a system prototype can be produced [Bratko, 1990].

Often efficiency improvements to logical programs can be achieved by changing the programming structure: the better ordering of clauses or procedures and goals; avoiding unnecessary backtracking or stopping the execution of useless alternatives as soon as possible; and changing the data structure representation.

Profilers have similarly been constructed for logical languages with a moderate amount of success. More recently profilers have been used to partition parallel Lisp programs so that effective mapping to a massively parallel architecture can be performed [Soden and Bock, 1995].

2.3.3 Functional Languages

Considerations for profiling

It is considerably more difficult to profile functional languages than it is to profile imperative languages. The difficulties can be categorised according to the following observations on the nature of the language:

1. Defining semantics to show the mapping of costs to program units — The abstract properties of lazy functional programs and the use of many small functions in programming means that it is not always clear where the resources involved in the evaluation of an expression should be attributed. This problem is reduced with semantic definitions which state that, given a certain specific computation there will be some definition of how the costs of evaluation will map to the different program units.

2. Ease of comprehension — It is not straightforward to predict the semantic behaviour in 1 (above), as predicting the behaviour of functional programs is difficult. Consider for instance the different space behaviour of the functions `foldl` and `foldr`. For functions such as `+` and `*`, that are strict in both arguments and can be computed in constant time, `foldl` is more efficient. For functions such as the logical 'and', `&&`, or the list operator `++`, that are non-strict in the first argument, `foldr` is more efficient [Bird and Wadler, 1988]. Making predictions about large programs on the basis of such observations is not a trivial task.

3. Difficulty of obtaining profiling information — Lazy functional languages, by their nature, are difficult to gather profiling information for. It is therefore

difficult to implement the specification presented in 1. Methods of compilation by transformation mean that the resulting executable code bears little resemblance to the structure of the original source code. Maintaining program units and allocating resource usage to these units becomes a difficult task.

These issues are supported by more specific language features which are listed below.

- Functional programs are designed to have -igh levels of abstraction; this means that the code produced resembles the problem specification. It is argued that this high level of abstraction makes coding easier, and indeed faster. Although it is a clear programming advantage, this high level of abstraction does mean that the executable form of a functional program is not a clear representation of the original source code. There are a number of difficulties that this causes in relation to efficiency analysis. Firstly, the extra levels of abstraction impose additional overheads when the program is run; secondly, the execution behaviour of the program is far less predictable. This issue is considered to support all three of the above categories.

- The explicit use of automatic memory management offers the programmer freedom from low-level issues such as the allocation of space for data structures. This method also introduces severe memory difficulties. The programmer is unaware of when memory is allocated and deallocated. Therefore, memory problems often only manifest themselves when the program runs out of heap space. Predicting how much memory space is used is difficult; the space taken by a list of 1000 integers may be a simple calculation in a language such as C, involving the number of bytes for each integer plus the number of bytes for the pointer or array structure in which it is stored. In a lazy functional language such a calculation is not possible, as the implementation of the list is reliant upon the implementation of lazy list constructors such as 'cons' operators as well as the integer representations themselves. This relates to program comprehension, the second category above.

- The use of higher-order functions offers another distinct programming advantage, though again this benefit is counterbalanced by the fact that higher-order functions cause profiling difficulties, as the function being applied may not be known at compile time. Rather, the function is passed as an argument or extracted from a data structure. This makes the task of attributing costs to the current section of code difficult. Consider also the definition '`f = foldr (+) [] xs`'. It is not easy to determine whether the costs of the function `+` should go to the function `foldr` or to the function `f`. This relates to 3, above.

- Functional programming advocates the use of many small functions; it is partly because of this that functional programs are described as easier to read than their imperative counterparts. Having profiling costs displayed in terms of thousands of small functions may not be useful to the programmer. Instead it may be more appropriate to display the results at a more meaningful part of the program. For example, where P_1, P_2, $P_3 ... P_n$ are sub-functions of an operation `parse`, rather than displaying profiling costs at the level of these sub-functions, it may be more appropriate to display the costs at the level of the parse operation. Methods of aggregating costs to parent functions mean that the mapping of costs to important parts of the code can be done with a reasonable amount of success (though this is discussed in more detail later). This relates to categories 1 and 3.

- Polymorphic functions such as `map` and `fold` encourage function reuse. These very basic functions can generate a large proportion of costs in a program by repeatedly applying the same function. It is not very useful however to know that your program spends 30% of its time in the function `map` as the `map` function will be used in a number of places in the program. What is needed in such a situation is the information which explains *which* calls to `map` cause the majority of these costs. This also relates to 1 and 3.

- Not all functional languages are lazy, but those which are offer additional problems. These are discussed in more detail below.

Lazy and Strict evaluation

In a strict language the way in which an expression is written, its source level definition, corresponds closely to its evaluation order at runtime. So for example[1] the expression:

$$x = f_{strict}\ exp_1, .., exp_n$$

would be evaluated as follows:

1. The arguments exp_1 to exp_n are evaluated independently of the definition of the function f_{strict}
2. The function f_{strict} and any sub-expression within it are evaluated
3. The result of this evaluation is bound to x

This scheme is termed *call by value.* The resources used by a function are easy to record. The result is the difference between the resources recorded at the start of the execution and those recorded at the end of the execution. There are no extra costs which occur after the execution of the function which should be included in the final results.

Lazy evaluation operates using an *call by need* scheme. Evaluation is delayed from the point where an expression is defined to the position in the code where its value is required. This is demonstrated by a further example:

$$y = f_{lazy}\ exp_1, .., exp_n$$

With lazy evaluation the order of evaluation of such an expression cannot be predicted without the definition of f_{lazy} and the context in which y is used. All that can be stated about the run-time behaviour is that the results of calling f_{lazy} will be bound to y. A function call may invoke evaluation of any of the arguments or may leave them unevaluated. The result may contain references to some of the arguments which may then be evaluated at a later time.

[1] This example is based on [Clayman, Clack and Parrott, 1995].

This problem is further complicated by the fact that a number of expressions may demand the results of a shared expression. Only the first function which calls the expression will bear the cost of evaluating the code. The remaining expressions get the result, in effect, for free. This presents a problem when profiling as there is no certainty as to exactly where the costs of the evaluation should be attributed. Dividing the costs between all the expressions which demand the results may seem like a sensible idea, though recording the statistics to be able to do this is difficult. For example, if

$$f_{lazy}\ arg_1, .., arg_n\ =\ (arg_1, arg_n)$$

in which the result is a pair containing the values of the first and the last arguments, further predictions can be made about the run-time behaviour. It is now certain that this pair can be evaluated without having to evaluate any of the further arguments. If the result y later appears in an expression of the form

$$z\ =\ (fst\ y)\ +\ (snd\ y)$$

then it can be demonstrated that the arguments exp_1 and exp_n will be needed in the evaluation of y and will therefore be evaluated. It is clearly not so easy to reason about exactly where expressions might be evaluated. In large lazy functional programs this problem becomes even more apparent as unevaluated objects may be passed through many function calls before evaluation occurs.

Different roles of programmers

A 'programmer' is a general term. Programmers may work at the machine code level, the operating system level or at an application level. There are many more distinctions which can be made, but this coarse characterisation is made to show that some programmers may find different profiling results more use than others.

This point is illustrated with an example presented by [Clayman, Clack and Parrott, 1995], demonstrating that a fundamental difference exists between the way in which applications programmers and system programmers (implementors) naturally reason about the execution costs of their programs. An application pro-

grammer's attention is focussed on the expense of particular expressions, especially those exhibiting degenerate behaviour. For example, in the following definition:

$$f\ g\ x\ =\ g\ expensive\ x$$

where g is non-strict in the first argument, lazy evaluation will cause *expensive* to be evaluated in g or in a function called from g (if it is evaluated at all). Strict evaluation however would have caused *expensive* to be evaluated in f, prior to calling g. If *expensive* is still treated as a lazy argument, but its execution time is attributed to f, then, should the *expensive* expression exhibit degenerate behaviour, the application programmer's attention will be drawn immediately to its declaration within f.

A system implementor may take a very different view of the situation. The actual behaviour of the program at runtime may be more important than the lexical relationship within the original source code. The effects of lazy evaluation must now be reported exactly as they occur, so that the knowledge can be used to improve the compile time and run-time heuristics (eg. dynamic scheduling.)

The different requirements and viewpoints of applications programmers and system implementors makes the connection between the definition of lazy functional source code and its eventual run-time behaviour a problematic task. Clayman and Clack describe this behaviour in stating that: the application programmer observes the run-time behaviour of the program and attempts to map this back to the original definitions in the source code; and the system implementor analyses the source program and attempts to predict its run-time behaviour. The run-time behaviour is also observed by the implementor in order to verify the success of the prediction. Analysis of the run-time behaviour is made with respect to the run-time domain, hence it is not necessary to map the run-time behaviour back to the lexical structure of the source code.

It is possible that the profiling behaviour of the systems implementor is different from the applications programmer, though from experience we believe that this is not necessarily true. Rather, the issue is not do with how these programmers profile their program, but at what level the profiling results are displayed to be of greatest

benefit to each programmer.

Application programmers are likely to find the results helpful if they are displayed at the level at which they program. Profiling results can be displayed in terms of the abstract data types or functions which he has written.

Programmers with a lower-level knowledge of the system, or with detailed knowledge of the language implementation may also find such results useful. They may also benefit from results which refer to a lower level in the programming language, such as numbers of closures, or heap constructors and producers.

The result of a function application may, in the first instance, be more useful if it is assigned to the unit of code in which the function is defined; in the second instance, the results may be more useful if they are assigned to the unit of code in which the function is evaluated.

Lexical vs Evaluation scoping

The difference in the examples above between the lazy and the strict case is significant. For a straightforward performance evaluation, applications programmers might find it simpler to reason about the behaviour of strict programs rather than that of lazy programs, as expressions are always evaluated by the functions in which they are declared.

To simplify the reasoning about lazy functional programs, the cost of evaluating expressions can be reported to the programmer with respect to the lexical structure of the source code. This method of program profiling is termed *lexical profiling.*

The difference between this method and the second form of program profiling called *dynamic or evaluation scoping* is subtle. With a strict language the two styles differ merely by the fact that one profiles functions and the other profiles expressions, but with a lazy language the differences are more fundamental. Lexical profiling measures whether work happens, and how much work happens, with the results being presented with respect to the source code. In contrast, evaluation profiling measures how much work is done and when the work is done.

In the example:

$$x = f_{strict}\ exp_1, .., exp_n$$

using a lexical profiling scheme, the costs of evaluating the arguments exp_1 to exp_n are attributed to the function x. In effect lexical scoping is similar to profiling a strict language; the difference is that only the cost of the actual evaluation performed on an argument is assigned to the enclosing function.

Using an evaluation-scoping profiling scheme, however, the results are very different. In the example:

$$y = f_{lazy}\ exp_1, .., exp_n$$

where

$$f_{lazy}\ arg_1, .., arg_n = (arg_1, arg_n)$$

and the function y is used in the definition,

$$z = (fst\ y) + (snd\ y)$$

the cost of evaluating the arguments arg_1 and arg_n are attributed to the function in which the evaluation took place, exactly when the work is done; the cost is attributed to the function z in this example.

In summary, lexical profiling assigns the costs of the evaluation of an expression to the function which built the expression; evaluation profiling assigns the cost of the evaluation of an expression to the first function which required its evaluation.

The following example programs illustrate the difference between the different profiling schemes. The examples, taken from [Clayman, Clack and Parrott, 1995], demonstrate the importance of understanding on which of the profiling methods the programmer's chosen profiler is based. A lazy evaluation scheme is assumed throughout.

	Function in which x is		Number of primitive operations					
			Lexical profile			Dynamic profile		
Program	declared	reduced	f	g	h	f	g	h
1	f	h	$1+P_x$	1	1	1	1	$1+P_x$
2	g	h	1	$1+P_x$	1	1	1	$1+P_x$
3	f	g	$1+P_x$	1	1	1	$1+P_x$	1

Figure 2.1: Number of primitive operations counted for the functions in each of the example programs.

Program 1

$$
\begin{aligned}
f \;&=\; (g\ x)\ /\ 18 \\
&\quad\; where\ x \;=\; < expression > \\
g\ x \;&=\; (h\ x)\ *\ 10 \\
h\ x \;&=\; x\ +\ 12
\end{aligned}
$$

Program 2

$$
\begin{aligned}
f \;&=\; (g\ 20)\ /\ 28 \\
g\ y \;&=\; (h\ x)\ *\ y \\
&\quad\; where\ x \;=\; < expression > \\
h\ x \;&=\; x\ +\ 22
\end{aligned}
$$

Program 3

$$
\begin{aligned}
f \;&=\; (g\ x)\ /\ 38 \\
&\quad\; where\ x \;=\; < expression > \\
g\ x \;&=\; x\ *\ (h\ 30) \\
h\ x \;&=\; x\ +\ 32
\end{aligned}
$$

The three programs above perform the same arithmetic operation but differ in the way in which x is declared and evaluated. Figure 2.1 contains a summary of the number of primitive operations counted for the functions in each program, using both the lexical and evaluation profiling schemes. The results of the lexical scheme always show the cost of x associated with the function in which x is lexically defined. The results of the dynamic scheme highlight the presence and effect of laziness, and the cost of x is shown associated with the function which required it.

The results are displayed not because one scheme is necessarily better than the other. More recently both schemes have been implemented within a single profiling tool; the user can then select either method depending on his requirements.

Graph reduction and profiling

Before a program is executed it is stored a computer's memory. There are many different ways that this program can be represented; a functional program is usually represented as a *graph.*

The goal of the execution of a functional program graph is to reduce the graph to *normal form,* printing the results as they become available. Most compilers perform this graph reduction by a method called *programmed graph reduction.* During the reduction graph nodes are consumed and detached from the graph; these nodes are replaced with new nodes until the graph has been completely reduced.

An example, taken from [Runciman and Wakeling, 1995], begins in Figure 2.2. This shows how the graph of the code, `sqr (5+2)`, is reduced to normal form. The `sqr` function is written `sqr x = x * x`. Functions and primitive operations serve as rewriting rules for parts of the graph. These rewriting rules can be implemented in machine code and will, when each rule is encountered, manipulate the graph accordingly.

Thus the technique has an internal representation and a 'machine' of instructions to manipulate this internal representation according to a certain reduction scheme. One implementation of this 'machine' is the G-machine, [Peyton Jones, 1987].

The G-machine code for the function `sqr` is:

```
sqr: PUSH 0
     PUSH 1
     PUSHFUN mul
     MKAP
     MKAP
     UPDATE 2
     POP 1
     UNWIND
```

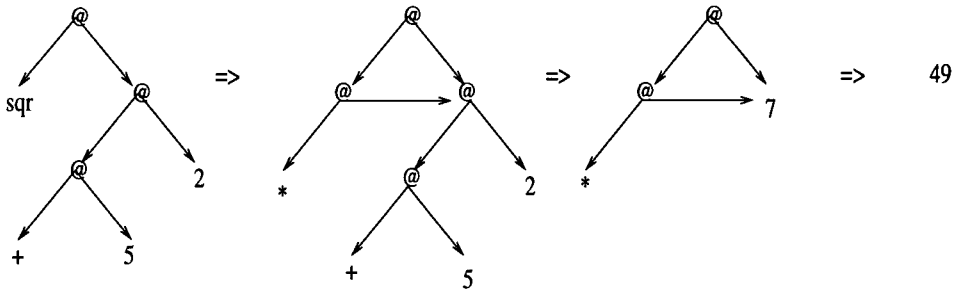


Figure 2.2: Graph reduction of `sqr (5 + 2)`.

Execution of the expression `sqr (5+2)`, seen in Figure 2.3, begins by pushing the pointer to the original graph onto the stack, part (i) of the figure. The 'spine' of the stack is then unwound by pushing pointers to the vertebrae of the spine onto the stack (ii). When the head of the spine is reached, the stack is rearranged so that efficient access to the arguments is possible (iii). The machine code instructions for the `sqr` function are then executed in sequence, until graph rewriting is no longer possible without the execution of individual functions (iv)-(xi).

Program graph reduction is an ideal mechanism for bridging the gap between abstract functional language programs and their low-level implementation. Many implementation ideas such as garbage collection and profiling can be based on the graph implementation and reduction mechanisms. However, the graph representation and transformation cause the representation of the program to move further away from the source code. This can often complicate relating resource costs back to the source program. This problem, together with the implementation of current profilers based on graph reduction is discussed in section 2.6.2.

2.4 Types of profiling tool

A closer look is taken at the types of profiling tools available. The first and simplest method of profiling to be discussed is occurrence profiling. Further categories of profiling tools correspond to the complexity measures discussed earlier; profilers which record timing behaviour are examined, as well as profilers which record memory usage of programs. There are many sub-categories of these methods, distinguished by the chosen method of data sampling.

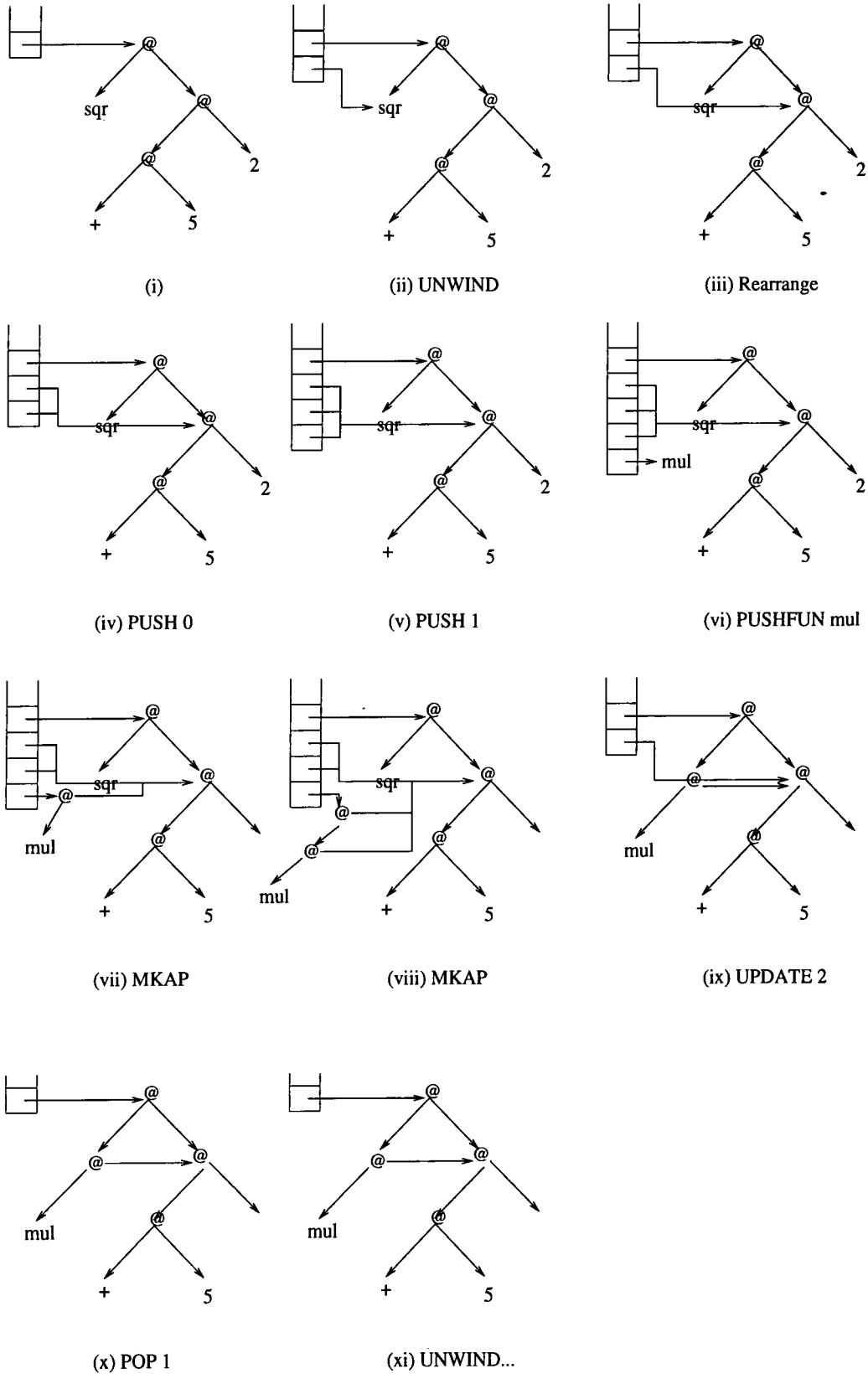


Figure 2.3: Programmed graph reduction of `sqr (5 + 2)`.

2.4.1 Occurrence profiling

A simple C program[2] which calculates prime numbers is considered.

```
        prime(n)
        int n;
        {
                int i;
                for (i=2; i<n; i++)
                     if (n % i == 0)
                         return 0;
                return 1;
        }

        main()
        {
                int i, n;
                n = 1000;
                for (i=2; i <= n; i++)
                     if (prime(i))
                         printf("%d\n", i);
        }
```

The simplest investigation into the behaviour of this program would be to consider the parts of the code which were called the most; the rate of occurrence at which sections of code were called. A simple occurrence profile may produce a copy of the code annotated with the number of times that code was called. For example:

```
        prime(n)
        int n;
        {
                int i;
999             for (i=2; i<n; i++)
78022                if (n % i == 0)
831                      return 0;
168             return 1;
        }
        main()
        {
                int i, n;
1               n = 1000;
1               for (i=2; i <= n; i++)
999                  if (prime(i))
168                      printf("%d\n", i);
        }
```

[2]and results, both of which are taken from the seminal paper by Bentley [Bentley, 1982].

The results tell us that the function `main` was called once and that it tested 999 integers with the function `prime`. By knowing that the function `prime` was called, it is possible to make predictions about what parts of the program have been executed and how many times these parts of the program have been visited, this is termed the *coverage* of the program. Coverage predictions about the above program show that its behaviour does correspond to the expected behaviour.

The program results can also be calibrated; number 1 was printed 168 times and the number 0, 831 times. It can be observed that most of the time was spent testing factors. The testing could be reduced to test only those factors up to the square root of `n`, rather than `n`. By changing the test accordingly and re-profiling, the results can be compared to decide which method is more satisfactory. This change to the program brings about a fourteen-fold improvement.

```
        root(n)
        int n;
5456    { return (int) sqrt((float) n); }
        prime(n)
        int n;
        {
                int i;
999             for (i=2; i<==root(n); i++)
5288                 if (n % i == 0)
831                      return 0;
168             return 1;
        }
        main()
        {
                int i, n;
1               n = 1000;
1               for (i=2; i <= n; i++)
999                  if (prime(i))
168                      printf("%d\n", i);
        }
```

Occurrence profiling is particularly effective for investigating the coverage of a program.

2.4.2 Time profiling

A more sophisticated method of profiling may record not only how often parts of the code were called but also the time spent in those parts of the program. Improving the occurrence count of sections of the code will often lead to improvements in performance. However, it may only be possible to reduce the occurrence count by performing a more complex calculation prior to calculating the relevant section of code, thus bringing the expensive computation forward in the order of the program evaluation. A time profile gives more insight into the CPU time needed by a program.

In the above example we may receive results such as:

%time	cumsecs	#call	ms/call	name
82.7	4.77			_sqrt
4.5	5.03	999	0.26	_prime
4.3	5.28	5456	0.05	_root
2.6	5.43			_frexp
1.4	5.51			__doprnt
1.2	5.57			__write
0.9	5.63			mcount
0.6	5.66			_creat
0.6	5.69			_printf
0.4	5.72	1	25.00	_main
0.3	5.73			_close
0.3	5.75			_exit
0.3	5.77			_isatty

The time is displayed in cumulative seconds and the procedures are listed in decreasing order of the percentage of execution time. Assumptions can be made about the time spent in each of these procedures and also how much time was spent performing activities such as output and housekeeping. The fourth column gives the time (in milli-seconds) spent performing one execution of the function. This may also be a useful measure, but it will later be demonstrated that this division of the percentage of execution time by the number of function calls can be misleading.

2.4.3 Allocation and heap profiling

Memory concerns in computing have almost gone a full circle. In the 1950s and 1960s programmers worked on small machines with a few kilobytes of memory. These machines contained a small number of instructions to allocate and deallocate parts of the computer's memory. The scarcity of the memory forced the programmers to develop neat and efficient ways to program, maximising the resources that they had in order to get the most computational power out of their machines.

By the 1970s memory had become an affordable commodity and as the price of thousands of kilobytes of memory dropped, so the size of software systems grew. During the following decade huge amounts of memory could be bought. The emphasis on programming changed from the 1960s paradigm of designing programs which were memory efficient, to the 1970s paradigm of producing orderly code, albeit in huge quantities. The literature of these times reflects these particular paradigms.

This view is changing once again as programmers have become more aware of the redundancies in their code and the unnecessary expense which is frequently incurred. There are often ways in which the memory consumption can be improved at little expense to the structure or the readability of the program code.

With this concern for memory use, there is a need to understand how memory is consumed by different parts of a program. A profile of the program's memory behaviour is required. There are many different ways in which to analyse this behaviour; these are categorised below:

Allocation Profiling — This is the simplest form of dynamic memory profiling; it records the number of memory cells allocated to the execution of a piece of code, even if these cells are only retained briefly. Each memory cell is marked by a label which indicates the code unit which allocated this part of memory. With these labels the number of cells allocated for a particular function can be determined. It is also possible to calculate the total number of memory cells allocated at any one

time by counting all of the cells with a label.

Labels can be recorded at each memory cell with only a small change to the underlying compiler. This method identifies those parts of the code which cause the allocation of large amounts of memory. This is of concern, but may be expected with large computations. Rather it is those memory cells which are allocated and then persist for a long time which are more worrying to the programmer and this information is not provided by the allocation profile. Detecting the source responsible for allocating the long-lasting data requires a more advanced profile called a heap profile.

Heap Profiling — Heap profiling is specific to functional languages. This periodically monitors the number of heap cells currently in use. Intermittent interrupts sample the heap every t seconds, where the value t can be specified at runtime. When the sample interval has passed and is detected by test code placed by the compiler at every code block, then the execution is suspended and the memory scanned. The cells in memory are tested to see if they are 'live' (currently being used by a part of the code). During this monitoring the profiling label associated with these live cells can be noted. These notes are later compiled into a results file. The profiling label can include information such as the cell's construction (what it represents in terms of symbols occurring in the source program) and its producer (which part of the program caused it to be produced). When the data is displayed as a final profile graph, the user can specify whether he wishes to have the heap identified as a constructor or a producer graph.

Separating the profiling overheads from the program data is easy, as although the system clock continues during these interrupts, this elapsed time is simply ignored. However, it was mentioned earlier that any method of sampling generated by external signals was likely to produce slightly different results each time the profiler was run. This heap profiling method is also prone to these sampling errors.

Heap Profiling (implemented with a Garbage Collector) — Functional programming languages offer a method of automatic memory management. At intervals in the program the execution will be interrupted, so that memory cells which are no

longer in use can be reclaimed. This process, termed *garbage collection*, can be combined with the heap profiling method, as garbage collection is effectively a natural break in the program's execution. When the cells in the heap are monitored for their longevity their profiling marker can also be identified. When the garbage collection is completed the results can be collected and written to a results file.

Heap Profiling (implemented with Garbage Collector and Timeouts) — It may be noted that garbage collection will not necessarily take place at regular intervals, so to avoid the samples taken during garbage collection being irregular, a timeout method is implemented which determines whether a garbage collection has taken place. If a garbage collection has not taken place a memory sweep will be done to sample the live cells on the heap and record the appropriate profiling information.

Leak Profiling — A memory leak is a general term given to the dynamic allocation of memory which is not reclaimed during the execution of a program. Leak profiles have been developed for C programs which identify heap objects which are never deallocated. A report on the call sequences responsible for allocating them is produced.

Lifetime Profiling — Recently a new method of heap profiling has emerged. This new method is based on the principle that heap cells are static; that is, the profiling labels associated with each cell remain fixed from the time the cell is allocated until it is garbage collected. Lifetime profiling records with the label the time in the execution at which the cell was created. With this information two attributes can be considered: the profile graph can display the heap objects broken down by the length of time each object lived, their *lifetime*; and a *retainer* profile displays the heap objects broken down by the set of producers which reference the object.

This information is intended to help the programmer identify not only 'who uses what memory' (the information that the regular heap profiler would supply), but also 'why the memory is retained' and what parts of the program are holding onto those parts of the memory.

2.5 Theoretical considerations for profiler design

2.5.1 Methods of Propagating Profile Times

There is a question in relating the measurement of resources to source code units of whether a cost of a function is simply its own cost or whether it includes the cost of any sub-functions which it calls.

If a function is to include the costs of a shared sub-function, then this issue is further complicated by the fact that the costs of the shared function must be accurately divided between the parent functions according to how costly each call was.

For a profiler to be completely accurate in its allocation of costs to source code units, it would have to be able to reconstruct the entire call path for all function calls, and store with this call path the costs incurred at this part of the program. This would enable the exact cost of shared sub-functions to be included in the costs of their parent functions.

Although this has been demonstrated to be practical for small example programs, on a larger scale this method has been dismissed as too costly.

In practice the information recorded at runtime is restricted to the calls made by a function to its immediate children [Graham, Kessler and Kusick, 1982]. For profiling tools this restriction presents a problem as they must estimate the execution time of more remote generations of sub-functions. For example, consider the following code:

$$
\begin{aligned}
f\ a &= h\ a\ +\ 1 \\
g\ b &= h\ b\ -\ 1 \\
h\ x &= i\ x\ +\ i\ x \\
i\ x &= x\ +\ 1
\end{aligned}
$$

and the call-graph segment for this code, see Figure 2.4, in which a function i is called by function h, which itself may be called by functions f or g.

It is uncertain exactly how much time is spent in or below function f, since the

function g may also call upon h, which itself calls upon function i.

One solution to this problem, which is termed the *statistical* approach, divides the time spent in function i according to the ratio of calls between functions f to h and functions g to h. For example if there are 8 calls from f to h and only 2 calls from g to h then the time spent in (or below) h will be divided 8:2, function f receiving the greater of these two figures. This method is unsatisfactory however as some calls to h may have a higher cost than others.

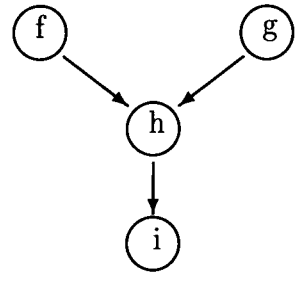


Figure 2.4: The program call-graph segment.

It is not always realistic to split costs according to the ratios based on the the number of times individual functions are called. For example:

$$
\begin{aligned}
f &= h\ 1000000 \\
g &= h\ 10 \\
h\ x &= factorial\ x
\end{aligned}
$$

Dividing the costs of the factorial function evenly between the two functions f and g would not be representative of the amount of resources used for each function call. A programmer would be more interested in the expensive call. In this example the distinction is easy to spot, but with hundreds of shared functions this task becomes extremely difficult.

An alternative solution is to allow the code for i to be subsumed by the code for h so that as far as profiling statistics are concerned i is an integral part of h; this is termed *inheritance* profiling[3]. This profiling method states that sub-function i

[3]or a *subsumed* profile if you read Sansom's thesis [Sansom, 1994]. It is important to note the distinction between this use of the term *inheritance*, forcing all shared functions to be treated as profiled functions and propagating any costs of the children of shared functions up to this part

is simply an extension of its parent, therefore the total amount of time spent in or below h due to f or g is determined absolutely; it is presented as a complete figure incorporating all costs for h and i.

Using this method it is difficult for a programmer to assimilate timing information for i or the profiled sub-functions of i, as i will seem to have no data relating to it—these results having been attributed directly to h. Even if the programmer is aware that this process is taking place the results can appear very misleading, particularly if most of the functions in the code are shared or are called by shared functions. Thus with inheritance profiling the code outlined would be profiled as though it had been written as

$$
\begin{aligned}
f\ a &= h\ a + 1 \\
g\ b &= h\ b - 1 \\
h\ x &= i\ x + i\ x \\
&\quad \textit{where} \\
&\quad i\ x = x + 1
\end{aligned}
$$

where the profiler only supplied results for those functions which were defined globally. With a more complicated call-graph many such functions may be inherited and the results may become restricted and misleading.

Inheritance profiling avoids the problems associated with shared children by refusing to propagate their time costs any higher than the shared child itself. This is achieved by forcing all shared children to be treated as profiled functions.

A profiler implementing inheritance profiling is still able to provide the equivalent of a statistical profiling technique. The profiling information can be presented statistically by post-processing the inheritance profile results. To achieve this all functions are profiled using inheritance mode, then the function call count information (which records not only how many times a function is called, but also the functions which called it) is used to statistically manipulate the results for the functions which the programmer wishes to profile. This does not entail the con-

of the code; and the use of *inheritance* adopted in this thesis, which describes the (accurate) subsuming of all costs to parent functions, whether the funtions are shared or not.

struction of a complete call-graph segment and is a relatively inexpensive way of providing statistical profiling.

Recursion groups

Functional programming adopts a style which is often recursive. The examples so far have concentrated on functions called in a serial manner; no mention has been made of recursion or mutual recursion (also referred to as a *cycle*).

Simple recursive functions should not cause a problem in the attribution of costs. A cycle on a single function can just have all the costs attributed to that particular function. The difficulty occurs with mutually recursive functions, as it can not be certain to which of the two (or more) functions the cost should be attributed.

One simple solution to this problem is to use static analysis on the program to collapse the costs of mutually recursive functions into one and record the results as such. Although this results in a loss of information, the results can show the relationship between the functions and therefore point towards this cycle in the code.

2.6 Current Profiling Tools

The discussion continues with some of the profilers for imperative languages which motivated the development of tools for functional languages. The examples from the imperative context are chosen because they represent developments in the theory, or because they are widely available and extensively used. All profilers for lazy functional languages are discussed.

2.6.1 Profilers for imperative languages

PROF 1982

One of the earlier UNIX profiling tools PROF produces an execution profile of a program by correlating the symbol table in the executable image file to a separate file containing execution time statistics [UNIX, 1979]. For each external symbol the percentage of time spent between that symbol and the next is printed in decreasing order, together with the number of milli-seconds per call. For example consider a simple Fibonacci program, shown in Figure 2.5.

```
int fib (x)
  int x;
{
  if ((x == 0) || (x == 1))
    return 1;
  else
    return ((fib (x-1)) + (fib (x-2)));
}

int main()
{
  printf("Fibonacci 25 is %d \n",fib(25));
}
```

Figure 2.5: A simple Fibonacci program written in C.

The program is defined as a recursive `fib` function and the function `main`; the results demonstrate (after some interpretation) that 65.6% of the time can be attributed to the function `fib`, results which are not considered unexpected in the circumstances. The remainder of the time can be attributed to the profiler itself; PROF does not use a scheme which separates the profiling costs from the rest of the program costs.

```
%Time Seconds Cumsecs  #Calls   msec/call  Name
 65.6    0.21    0.21  242785      0.0009  fib
 34.4    0.11    0.32                      _mcount
  0.0    0.00    0.32       1      0.      main
  0.0    0.00    0.32       4      0.      atexit
  0.0    0.00    0.32       1      0.      _profil
  0.0    0.00    0.32       1      0.      printf
```

The monitoring of program symbols by the PROF profiler must be done with care as sometimes the static functions are not correctly located and the profiling results are attributed to the wrong function.

The profiling times reported from identical runs of the PROF profiler on a program may be different because of the varying cache-hit ratios that result from sharing the cache with other processes. This may happen even if the program seems to be the only one using the machine. Hidden background processes may still distort the data. Occasionally the system clock ticks will cause the profiler to 'beat' in program loops. In such cases the results will be greatly misleading.

No attempt is made in the profiler to split the cost of shared functions between the parent functions. Indeed no attempt is made to show the inheritance of any of the costs from lower level functions in the call-graph to the higher level functions. Changing the Fibonacci program above to include two functions `a` and `b` which separately call the `fib` function with the arguments 25 and 5 respectively produces the following PROF results.

```
%Time Seconds Cumsecs  #Calls   msec/call  Name
 66.7    0.18    0.18  242800      0.0007  fib
 33.3    0.09    0.27                      _mcount
  0.0    0.00    0.27       1      0.      a
  0.0    0.00    0.27       1      0.      b
  0.0    0.00    0.27       1      0.      main
```

No information is recorded about what call to the `fib` function is the most expensive. The profile information still remains useful but the programmer should be aware of the very many restrictions imposed in its collection.

The profiler can be categorised as follows:

- The profiling results are presented in a tabular form. A simple plot of the same results can also be requested.
- The results have no form of inheritance. They are 'flat' profiles.
- PROF only measures time profiles and makes no attempt to record any information about memory behaviour.
- The resource usage is recorded by the execution sampling method described earlier.

GPROF (and MPROF) 1982

A similar UNIX tool GPROF [Graham, Kessler and Kusick, 1982] produces a profile of a program based upon the call-graph of a program's execution. Results are presented with an entry for each function together with its call-graph parents and call-graph children. The execution profile of a program is again produced by correlating the symbol table in the executable image file with a call-graph profile file in a two stage process.

Initially execution times for each routine are propagated along the edges of the call graph. Cycles are discovered and calls into a cycle are made to share the time of the cycle.

A GPROF profile is taken of the extended Fibonacci program. The first listing shows the functions sorted according to the time they represent, including the time of their call-graph descendants. Below each function entry is shown its direct call-graph children and how their times are propagated to this function. A similar display above the function shows how this function time and the time of the descendants are propagated to its direct call-graph parents. This profile information is described as a *call-graph profile.*

```
granularity: each sample hit covers 4 byte(s) for 2.22% of 0.45 secs
                                     called/total       parents
index  %time    self descendants  called+self    name           index
                                     called/total       children
[1]     64.4    0.29        0.00                 internal_mcount [1]
-----------------------------------------------
                                    242798              fib [2]
                0.08        0.00        1/2             a   [5]
                0.08        0.00        1/2             b   [6]
[2]     35.6    0.16        0.00        2+242798    fib [2]
                                    242798              fib [2]
-----------------------------------------------
                0.00        0.16        1/1             _start [4]
[3]     35.6    0.00        0.16        1           main  [3]
                0.00        0.08        1/1             a [5]
                0.00        0.08        1/1             b [6]
-----------------------------------------------
[4]     35.6    0.00        0.16                    _start   [4]
                0.00        0.16        1/1             main [3]
-----------------------------------------------
                0.00        0.08        1/1             main [3]
[5]     17.8    0.00        0.08        1           a [5]
                0.08        0.00        1/2             fib  [2]

-----------------------------------------------
                0.00        0.08        1/1             main [3]
[6]     17.8    0.00        0.08        1           b [6]
                0.08        0.00        1/2             fib  [2]
-----------------------------------------------
```

GPROF employs a statistical method of passing on the inherited costs of shared functions to the parent functions. The results above show that the cost of the `fib` function are split evenly between the functions `a` and `b`; each is given 17.8% of the costs. In this case the profiling results simply do not represent the resources used by the functions of the Fibonacci program. The programmer must be aware that quantisation errors in GPROF profiling are rife.

Another problem associated with GPROF is that it does not monitor space usage and so cannot provide complete information for programs which make full use of dynamic memory allocation. Finally it is noted that GPROF does not provide useful information for mutually recursive functions, as it collapses each strongly connected component in the syntax graph to a single point.

A later experimental version of this tool named MPROF [Zorn and Hilfinger, 1988] attempted to attribute memory allocation data to all the parents of functions up to a depth of 5. The results of the MPROF tool were recorded in the same manner as those of the GPROF profiler; the overheads of such a scheme were therefore enormously high which finally meant that it was not possible to develop this idea in any more detail. As part of the research in this thesis we return to this idea, but within the scope of an efficient implementation; see chapter 5.

GPROF, despite its difficulties, has motivated a great deal of the work on the profiling of lazy functional languages and has itself been a useful tool to programmers of imperative languages. .

The profiler can be categorised as follows:

- The profiling results are presented in a tabular form. A simple plot of the same results can also be requested.
- The profiling results are presented as 'call-graph' profiles and 'flat' profiles.
- GPROF only measures time profiles and makes no attempt to record any information about memory behaviour (although MPROF does profile memory consumption).
- The resource usage is recorded by the execution sampling method.
- Statistical inheritance is used to pass the inherited costs of shared functions to their parents.

TCOV 1988 and PROFIL 1990

While discussing imperative profiling tools available under the UNIX operating system it is worth noting the tool TCOV (Test COVerage analysis) which produces a statement by statement occurrence profile, prefixing the statements with the number of times they have been executed. The profile produced however includes only the number of times each statement was executed, not the execution times themselves [UNIX, 1979].

This profiling tool:

- Provides occurrence counts of programs.
- Provides no time or memory profiles.
- Presents its results by annotating the source code.
- Uses a procedure timing method of producing the occurrence profiling results.

PROFIL produces run-time execution profiles based on timing behaviour. This profiler uses a procedure timing method of collecting its profiling results. PROFIL is implemented as a function which samples the program counter at set intervals. At each interval the program counter is recorded so that the timing results can be related to the part of the program currently executing.

PROFIL:

- Provides time profiling based on 'flat' profiles.
- Presents results by annotating the source code.
- Uses a procedure timing method of producing time profiling results.

Both methods introduce statistical coverage at a very low level which, though useful for systems programmers, provides little help to applications programmers working on large-scale systems.

2.6.2 Profilers for functional languages

Statistics Provided by Functional Language Implementations

Many functional language implementations (eg. Miranda [Turner, 1985]) provide simple statistical information dealing with such things as the number of reductions performed and the total number of heap cells allocated during an evaluation. These can be used to choose between different program designs; if we wish to determine

which of two programs is more efficient we can compare their run-time behaviour for a given input.

The execution of a simple Miranda program produces the following statistical information.

```
501
reductions = 12027, cells claimed = 20035, no gc's = 2, cpu = 1.77
```

These measures produce useful comparative statistics but they do not correspond directly to the amount of resource used. The number of reductions recorded can be a deceptive figure in the analysis of a program's behaviour, as it is closely tied to the implementation of the language. The number of cells claimed and the number of garbage collections do provide useful information about reclaimed memory, but they do not say exactly how much memory is used. These statistics are basic and provide limited resource analysis.

The New Jersey SML Profiler, 1988

The New Jersey version of Standard ML [Appel, Duba and MacQueen, 1988] is noted for the fact that it also supplied a profiler which extended the basic statistical information to include results about individual functions and not just the program as a whole.

The profiling system works by the programmer choosing a subset of functions to profile. The children of these functions, for the purpose of profiling, are subsumed into their parents, the method previously described as the *inheritance* profiling technique.

The New Jersey profiler only works for strict evaluation and makes no attempt to profile heap space. The results themselves are not completely clear because of a number of schemes adopted to produce the results. For instance higher-order functions have the costs of all their arguments attributed to special identifiers rather than the actual function names. Tolmach and Dingle (who developed debuggers

of SML) state that having these slightly ambiguous results should make little difference to the interpretation of the results of small programs, where a higher-order function may only be called once or twice. Guessing to which function the special name refers may be simple for smaller programs, though performing this mapping is not so easy for larger programs.

It is suggested in the profiler's accompanying manual that the programmer select different groups of functions during a collection of profile experiments in order to produce an accurate picture of the results. Changing these groups of selected functions is a useful idea when profiling; often the programmer will prefer to see parts of the results rather than all of the results at one time. Re-running the profiling experiments again and again to collect these different results can be a timely task, particularly if the program being profiled is large.

The profiler falls into the following categories:

- Timed profiling results are produced. No memory profile is available.
- The results are produced in tabular form.
- The profiler uses an inheritance approach to subsume the costs of sub-functions.

Inline Cost Primitive Profiler, 1990

One of the earlier attempts at profiling lazy functional programs began at UCL [Parrott and Clayman, 1990] with the introduction of inline cost functions. Using the cost function, which is similar in behaviour to the identity function, it is possible for a function to measure the cost of evaluating its own arguments. This is achieved using evaluation transformers and forcing the correct amount of evaluation for the particular argument to occur inside the cost function. The results are written to a special output stream which cannot be accessed by the program.

For such a primitive to work properly it was necessary to ensure that the evaluation of expressions took place at the right time for the results to be meaningful. Normally an application of the identity function would not cause its arguments to

be evaluated. However, it is desirable that the argument of the cost function is evaluated to the extent demanded by the context of the application occurrence. An example is taken from Clayman, Parrott and Clack's paper [Clayman, Parrott and Clack, 1991]. In the expression:

$$n + fst\ (cost\ expr)$$

expr is a pair whose first element is a number. In this context the cost to be measured is that of constructing the pair and evaluating the first element. To force the evaluation to occur inside the *cost* primitive, a method is used which is based on Burn's evaluation transformers [Burn, 1987].

The fundamental problem with this profiling technique is that it takes a 'microscopic' view of the program. The information supplied by the cost function is dependent on its context at runtime, making the results very difficult to interpret unless the full effects of laziness are understood. Furthermore, the cost functions do not provide information about space usage or function call-counts.

The Inline Cost Primitive Profiler:

- Produces timing information for lazy functional programs written in the intermediate language FLIC.
- Uses a procedure timing method of producing time profiling results.

This work was a prelude to the UCL Lexical Profiler, below.

The UCL Lexical Profiler, 1991

The UCL Lexical Profiler [Clayman, Parrott and Clack, 1991] is a tool for profiling lazy, higher-order functional programs. This profiler introduced the lexical profiling technique described in section 2.3.3, providing profiling information related to the way in which the program was written rather than the way in which the program was evaluated. The profiler records time information, occurrence counts and also heap residency.

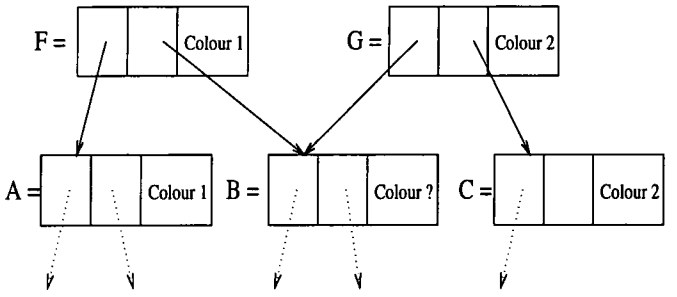


Figure 2.6: An example of an un-profiled function which is shared by two profiled functions.

To relate the resource information back to the source code program, each function which is profiled is given a unique label termed a 'colour'. This label is assigned the time and space costs of those expressions defined within the lexical scope of the function.

Each program in the UCL compiler is represented internally as a parse-graph, similar to the graph form seen in section 2.3.3. To keep track of the association between the parse-graph as it is transformed and the source program, each of the graph nodes is assigned a colour from its source function. A colour is typically a unique integer.

As the high-level function names are not stored in the internal parse-graph, a table is collated containing the colours and the source code functions to which these refer. This then means that the profiling information can be mapped back to the original source code.

The colours are propagated to other areas of the graph not assigned a colour so that all the parts of the graph have an associated colour. The costs of reducing all the parts of the graph can then be assigned to these colours.

Unfortunately, propagating these colours to other areas of the graph can cause potential problems. For example, Figure 2.6 illustrates the problem of an un-profiled shared function shared by two (or more) profiled functions.

To overcome this problem the UCL profiler requires all shared functions to be profiled separately (the inheritance profiling technique); this then prevents the costs of shared functions from being inherited incorrectly. This is clearly a major

drawback with the scheme as many of the costs incurred by a program would be due to the use of low-level shared functions. By preventing these costs from being propogated up the call-graph it is very difficult for programmers to get a higher level profile of their program.

To some extent this problem was recognised and in response a method of profiled pairs was suggested. By storing the colour of the function and the colour of its origin (its parent function), run-time pairs could be recorded. These pairs were proposed to form the basis of statistical inheritance for the profiler. This would have gone some way towards solving this problem, however it was never implemented.

The UCL profiler uses the procedure timing method of profiling, where instructions inserted into the program code initiate sampling of the system clock. There are two principal drawbacks with this scheme (introduced in section 2.2.2); firstly, this method is an expensive, time-consuming approach and secondly, the accuracy of the sampling method is based upon the accuracy of the system clock. These two problems will be exacerbated by the evaluation scheme. Under a lazy evaluation scheme there will be a large amount of interleaving between evaluating functions and therefore the elapsed intervals will often be very short.

The authors state in [Clayman, Clack and Parrott, 1995] that the prototype was not extended to allow the profiling of standard functional source code. As it stands the profiler uses FLIC code [Peyton Jones and Joy, 1989] as its input; although Haskell code can be profiled, it must first be converted into this FLIC intermediate code.

In summary:

- Occurrence, timed and heap residency profiling results are produced by the Lexical profiler. These are displayed in graphical and tabular form.
- The results are collected by a procedure sampling method.
- The propagation of results is done by inheritance profiling described in section 2.5. No form of statistical inheritance is adopted.

- The functional program must be converted to FLIC code before it can be profiled.

Heap Profiling, 1992

Runciman and Wakeling [Runciman and Wakeling, 1992a] describe a serial profiler which monitors heap usage (residency) of lazy functional programs, but which does not measure other factors such as call counts or the time spent in functions.

Their method similarly uses a graph reduction technique. When the execution of a particular functional program requires large amounts of memory, it is possible to sample which nodes (or which type of node) in the graph occupy the most space for the longest time, and which functions were the immediate cause of those nodes being introduced into the graph.

The first part of the tool is a modified compiler which attaches two tags to every cell in the heap, identifying the function that produced the graph node and the constructor that the graph represents. The second component of the tool is a program which displays the heap profiling statistics in the form of a graph.

The presentation of the results of the profiler is impressive, but the product of (time * space), used as a measure of the cost of the whole program, is deliberately not analysed with respect to individual functions.

With experience a programmer can, when using these profile graphs, successfully isolate critical definitions which may be using disproportionate amounts of heap space. Runciman and Wakeling have shown gains of factors of 2 or 3 on large programs, impressive results by any means. However, the user must understand how the run-time system works. At the current level of implementation it is necessary for the user to display a wider knowledge of the underlying implementation than would be expected of a typical applications programmer.

The Runciman and Wakeling profiler is undoubtedly a major step forward, but, as Clayman [Clayman, Clack and Parrott, 1995] points out, it is deficient in a number of areas:

- The producer profile can indicate that certain functions (eg. `map`) are responsible for producing disproportionate amounts of cells, but there may be many applications of the function in a program and the producer profile cannot distinguish between the different applications; the profiler does not produce any method of aggregation for these low-level functions.
- There is (deliberately) no information about the time spent in functions or the number of times each function is called.
- The profiling statistics are calculated by visiting the program-graph at predetermined intervals, the results of which are displayed as heap consumption in bytes. For large heaps the delay caused by these visits to the call-graph may be long, thus, for practical reasons, an upper-bound is imposed upon the sample frequency. This may have an adverse effect on the results produced, since the changes in the sample frequencies can cause profiled data to be inaccurate.

In summary:

- The profiler produces serial heap profiles of Haskell programs compiled with HBC (the Haskell 'B' compiler).
- The results are deliberately displayed at a low level. It may therefore be necessary to have a wider knowledge of the programming language and implementation than the average applications programmer.
- The results are displayed in a PostScript graph. The graph can be selected to display a constructor or a producer profile.
- The resource usage is recorded by the execution sampling method.

nhc heap profiler

nhc (*n*early a *h*askell *c*ompiler) [Runciman and Röjemo, 1996] is a subset of Haskell written for a machine with a small amount of memory (less than 4Mb). It is

based on the HBC/LML compiler and provides only the essentials of a G-machine implementation.

Two profilers are implemented on this compiler; the first is a lifetime profile which shows dynamic information about heap cells. Each heap cell is marked with its creation time, which allows a post-processor to determine the age of the cells. The second profiler is a retainer profile. This begins to explore not only the producers of heap cells, but also the consumers or retainers. This then allows the user to see not only what is on the heap but also what is retaining these objects[4].

Our experience of using the Runciman and Wakeling heap profiler has demonstrated a number of occasions where such a profiling method would have been very useful; see case study II of chapter 4. The retainer profile has potential to resolve many of the space leaks experienced in the development of lazy functional programmed systems. It is thought that this approach will offer a promising insight into heap profiling.

'Cost-Centre' Profiling, 1994

The 'cost-centre' profiler [Sansom and Peyton Jones, 1994,1995][Sansom, 1994] specifically addresses the problem of attributing data gathered during the program's execution back to the source code. Distributed with the Glasgow Haskell Compiler [Peyton Jones, Hall and Hammond, 1993] the profiler offers occurrence counts, timed and heap profiling, including serial profiling of time and memory usage.

The source code is annotated either by hand or automatically with cost centres, labels to which the costs of executing the enclosed expression are attributed. Associating the cost centre with an expression is made explicit by extending the syntax of a Haskell expression with an `scc` (set cost centre) construct.

Operationally the `scc` expression attributes the cost of evaluating the expression to the cost-centre label (which is either a user specifiable string or a function name). Semantically however, an `scc` expression simply returns the value of an expression.

[4]These methods of profiling were described in section 2.4.3.

scc expressions are scoped so that

```
mapg x l = (scc "mapping" map expr) (g x) l
```

will attribute the cost of evaluating the map, and any non-profiled function below that, to the label "mapping". Any other costs will be attributed to another cost centre, which perhaps encloses the function mapg for instance[5].

All the instances of evaluating an scc-annotated expression are lexically (or dynamically, if specified by the user) attributed to its cost centre. Any un-profiled expressions are subsumed into the surrounding cost centre. There is however no method of inheriting the cost of profiled sub-functions; costs are only attributed to a single cost centre and the results produced are flat profiles.

The profiler is implemented as an integral part of the Glasgow Haskell Compiler (GHC). An scc expression can be included in the original source of the Haskell code by the programmer. Alternatively these scc expressions can be automatically included in the code by selecting the option to profile all global functions. Every pass of the compiler had to be extended to account for this extension to the Haskell language.

GHC is an optimising compiler which transforms the code during compilation to an intermediate Core language and finally to a resulting program graph, section 2.3.3. An important part of the cost-centre profiler is that it preserves the scope of the cost centres during these program transformations so that evaluation is not moved from the scope of one cost centre to the scope of the next. The code generated by the compiler is based on the Spineless Tagless G-machine model of graph reduction [Peyton Jones, 1992].

Recording the costs during program execution is not difficult. Three STG-level extensions are implemented to enable run-time profiling: A *current cost centre* is added to the machine state, this current cost centre will continuously change throughout the execution of the program and any execution costs are attributed to

[5]The reader will note that the cost-centre scheme is similar to that of the profiling colours described in the UCL Lexical profiler.

this current cost centre; all heap closures have a cost centre attached to them, so whenever a closure is entered its cost centre is loaded into the current cost centre; during program evaluation enclosing cost centres are stored on the return stack, these are restored whenever the constructor is returned, and the associated code continues to be evaluated in the scope of the correctly enclosing cost centre.

This system has had a large amount of success and produced a number of improvements to the profiling of large systems, [Sansom, 1994][Morgan, Garigliano, Jarvis and Parker, 1994]. The cost-centre profiler:

- Profiles Haskell programs to produce flat results in a tabular, graphical and serial time manner.
- Profiles programs for occurrence, time and heap usage.
- Records the resource usage by the execution sampling method.

2.7 Chapter summary

This chapter has introduced literature and previous research concerning the efficiency analysis of programs. In particular the chapter has presented issues related to practical methods of measuring resource usage in programs. Practical methods may be more useful in the analysis of large programs as their measurements can deal with average-case analysis which is often a more useful means of evaluation.

It is necessary to consider a number of issues when designing practical complexity analysis tools. For example, it is important to carefully select what will be measured by the tool, how these metrics will be recorded and then how they will be presented in the final results to the programmer. Relating the resource usage back to the original source code is a non-trivial task.

Profiling tools have to overcome a number of specific design issues related to different programming paradigms. Profilers for imperative and logical languages are easier to construct as the evaluation of programs written in each of these paradigms

follows a strict order. Resource usage can be measured between two points in the program to determine the overheads for individual parts of the code.

Profiling functional programs, especially those which use lazy evaluation, is more difficult. Functional programs are abstract, allowing the programmer to construct code which bears little resemblance to the underlying machine representation and evaluation mechanisms. The problems of profiling lazy functional programs were discussed.

The different types of profiling tools were described. These included occurrence profiling, timed profiling and memory profiling.

A theoretical design issue in the development of profiling tools is whether to include in the cost of functions the costs of any sub-functions. If the costs of sub-functions are not included, the results may be difficult to interpret as they may only be displayed in terms of low-level functions. This is described as a flat profile. Some profilers attempt to inherit the results of profiled sub-functions by differing means of subsuming costs. If the program includes shared functions then compromises are made about how the results are recorded or how they are inherited to parent functions.

The chapter ended by presenting some of the more common profilers for imperative languages, some profilers for strict functional languages and finally all of the profilers for lazy functional languages. These were arranged in the theoretical categories into which they fall and their benefits and drawbacks were discussed.

Chapter 3

Large-Scale Functional Systems

3.1 Introduction

There are aspects of large programmed systems which make them fundamentally different from small programs. As a system increases in size, the programmer is forced to respond to a number of conditions; the evolving program changes and its structure becomes more complex unless active efforts are made to avoid this phenomenon (the law of *increasing complexity* [1]).

If the program is used in a real-world environment then it must change in order to become progressively useful in that environment (the law of *continuing change*). The growth and continued change will often be a self regulating process and the measurement of system attributes such as size, time between releases and the number of reported errors reveals significant trends and invariances (law of *program evolution*).

The rate of change is governed by the amount of resources devoted to system development, (law of *organisational stability*) and over the lifetime of the system the incremental change in each release is approximately constant (law of *conservation of familiarity*).

[1] of *program evolution dynamics* [Sommerville, 1992].

The programming style will change as the system evolves. A programming language is more than just a means for instructing a computer to perform tasks; the language also serves as a framework within which we organise our ideas about processes. Thus when a language is described, particular attention should be paid to the means provided by the language for combining simple ideas to form more complex ideas. Every powerful language has three mechanisms for accomplishing this:

- *primitive expressions*, which represent the simplest entities with which the language is concerned,
- *means of combination*, by which compound expressions are built from simpler ones, and
- *means of abstraction*, by which compound objects can be named and manipulated as units.

These methods will be employed by programmers to progressively build up the systematic structure of the developing project. Different programmers may develop different methods in order to implement their abstract levels, separated from, and utilised by, another part of the code by means of abstraction barriers.

The programmers of a system will also change as a result of the increasing development. Different levels of expertise will be needed by different parts of the system, from the knowledge of low-level implementation details to higher-level application details. These programmers may have little to do with the others' code, in some cases being completely separated from other programmers by progressive levels of code detail.

This chapter deals with the development of large-scale functional systems. In particular the chapter is based on the development of the LOLITA system at the University of Durham.

The Laboratory for Natural Language Engineering at the University of Durham has developed the LOLITA system for natural language processing applications

[Garigliano, Morgan and Smith, 1992,1993][Garigliano and Long, 1994] [Jarvis, Poria and Morgan, 1995][Morgan, Garigliano, Jarvis and Parker, 1994, 1996]. The system consists of 50,000 lines of source code (not including comments; with comments, the system consists of approximately 80,000 lines), equivalent to about 500,000 lines of imperative code [Turner, 1982], divided between 170 modules. In addition there are around 450 data files. Although the system was initially developed by one person, a team of approximately twenty people is currently engaged in developing various aspects of LOLITA. The LOLITA system is written entirely in Haskell.

The LOLITA system may therefore be described as a large-scale program within the functional programming community. Traditionally the description of large-scale systems has focussed around developments based upon imperative languages, but with the more recent development of large-scale functional systems this field can be extended. There has been a recent drive to identify real-world functional systems (see next section), though little has been done to explore the construction of these systems on a larger scale. This chapter endeavours to explore the issue of large, real-world functional programs. It is the work on a large-scale program and the investigation of the principal differences between large- and small-scale functional programs which promotes this thesis.

3.2 Functional Programming at Large

3.2.1 Real-World systems

There are simple criteria for systems being classed as 'Real-World', that is they must be written primarily to perform a task, rather than to experiment with a programming language.

In May 1994, Giegerich and Hughes organised a successful workshop on Functional Programming in the Real World [Gill and Wadler, 1995]. It was identified that there were a number of real-world applications of functional programming.

Twenty-three different real-world systems were identified at the workshop and described in some detail by their representatives. Of these twenty-three systems eleven were written in some form of ML [Milner, 1990].

ML, which stands for Meta-Language, is a family of advanced programming languages with functional control structures, a polymorphic type system and parameterised modules. ML has strict semantics although there are several implementations of ML including the Standard ML (SML) and also a version with lazy semantics (LML). ML is a well-established and well-documented language, it contains a supportive tool-kit and has users both in industry and academia. Of the eleven cited real-world systems written in ML, ten were implemented in a strict version of the language. Seven of these real-world systems had been developed at Carnegie Mellon University in the US.

One of the more prominent examples of a real-world system written entirely in ML is the Isabelle generic theorem prover, a 16,000 line SML program, written in 1986 by Larry Paulson of the University of Cambridge [Paulson, 1986].

Ten of the real-world systems presented were written in Haskell. The Haskell language is the result of a language design committee set up in 1987 to prevent the spread of a collection of non-standard, non-strict, purely functional programming languages, a problem which was facing the community in the mid-1980s [Hudak and Fasel, 1992]. By ML standards Haskell is a young language and the fact that ten recognised real-world systems have been written in Haskell testifies to its success. Five of these systems however (Anonymous FTP Client, Cherry, Equational Reasoning Assistant, Happy and Network Tool-kit for Haskell) are written by a single team, Andrew Gill and Darren Moffat from Glasgow University in the UK. The Glasgow Haskell Compiler (GHC) is also cited as one of the real-world functional systems written in Haskell. Though a compiler is clearly a real-world system, there is some debate as to whether a compiler should be classed in this discussion alongside systems such as LOLITA for example. Rather, there should possibly be a distinction between those systems which are written by people with a vested interest in the use of functional languages and those written by people who simply

choose a functional language because it is suitable for a certain problem, or simply as an arbitrary choice. This distinction is made by redefining real-world systems to include two groups:

- *Related Real-World Systems*[2] — the system designers and writers have a particular interest and expertise in functional programming languages. The development of the system in a specific programming language may be motivated by this interest.
- *Unrelated Real-World Systems* — written primarily to perform a task, rather than to experiment with functional programming languages. The choice of a functional programming language reflects the task at hand (e.g., mathematical or rule-based), or is arbitrary. A large amount of the expertise in functional programming is built up during development.

The Standard ML compiler of New Jersey and the Haskell Compiler (HBC) of Chalmers University would, with GHC, fall into the first category. The three remaining Haskell systems are the Mitre Speech Recognition System [Goblirsch, 1993], Ebnf2ps railroad diagram drawing tool from Tübingen [Thiemann, 1994] and the LOLITA system from Durham; these should fall into the latter category.

The remaining two systems presented at the Dagstuhl workshop were written in Miranda, a non-strict, polymorphically-typed purely functional language. The first of these systems, MC-SYM, computes the 3D shape of a piece of nucleic acid molecule given the sequence of nucleotides and a set of constraints to satisfy. This system, from the University of Montreal [Major, Lapalme and Cedergren, 1991], has also been implemented in Scheme, Multilisp, and C. Finally, Smart-Card, from the University of Amsterdam, is a prototype of a smart-card operating system [Hartel and de Jong, 1994].

The Dagstuhl workshop highlighted a number of real-world systems, although analysis of these systems shows that there are less unique examples of real-world systems than it first appears:

[2]Clearly this choice of terminology is flexible.

Development Centre	Implementation Language Used			
	ML	Haskell	Miranda	Other
Glasgow University		6		
CMU	7			
Chalmers University	1	1		
Other	3	3	2	

Figure 3.1: Summary of the results of the Dagstuhl workshop on Functional Programming in the Real World.

- three of the twenty-three systems are functional programming compilers and may be more accurately categorised as Related Real-World Systems;
- thirteen systems had been developed at Glasgow or Carnegie Mellon universities;
- twenty-one of the systems had been written in either Haskell or ML (or a ML derivative), the remaining two systems are written in Miranda;
- eleven systems are written in a strict functional language, twelve are written in a lazy language; of the twelve lazy systems only five are not written at Glasgow or Chalmers Universities.

These results are summarised in Figure 3.1.

The results of the Dagstuhl workshop are not complete; there are many other systems programmed in functional languages which were not represented at the workshop. In particular the language Erlang has been used in software development projects at Ericsson since 1990, [Armstrong, Williams and Verding, 1993]. Erlang is a concurrent functional programming language which has a significant use in the development of large industrial real-time systems. The language, though not strictly functional, is untyped and uses a pattern matching syntax. It also utilises recursion equations, explicit concurrency and asynchronous message passing. It is

relatively free from side-effects. This language offers 'functional properties' and the experience gained in the development of many substantial projects may yet prove to be valuable in the development of similar 'purely functional' systems.

The recent book by Runciman and Wakeling [Runciman and Wakeling, 1995] discusses some applications of functional programming. These applications were developed as part of the FLARE project, collaborative work within the UK's Information Engineering Advanced Technology Programme, whose aim was to put functional programming into the hands of potential users of real applications. One notable application is the Veritas proof assistant [Hanna, Daeche and Howells, 1992]. Implemented in both SML (12,500 lines) and Haskell (11,500 lines), the project has provided useful information on the efficiency (in space and time) of Haskell programs. It also considers at great length the problems of error handling in functional programs and provides some effective alternatives to the exception handling mechanism.

Despite the encouraging results recorded in the Runciman and Wakeling book and other texts, the results of the Dagstuhl workshop do point to a surprisingly small number of Unrelated Real-World Systems developed independently of the main universities established as functional programming research centres.

The LOLITA system is an Unrelated Real-World System written in a lazy functional programming language. There are few examples of such a system and the results of the Dagstuhl workshop would suggest that the experiences of developing the LOLITA system are perhaps unique. Its scale and the period of time over which it has been developed have given a large amount of experience and insight which is elaborated upon throughout this chapter.

3.2.2 Program comprehension

Most people accept that the choice of an ideal programming language for a certain task depends greatly on the nature of that task. There is therefore no notion of a universal programming language which is well suited to all programming tasks; the

vast number of languages in existence is evidence of that. There must, therefore, be features of functional programming languages which make them suitable for system development, features which will make the software engineer adopt a functional programming language rather than an imperative language when he codes his system.

Despite the relative lack of unrelated real-world systems, it has often been argued that it is easier to write in a functional programming language than in an imperative language. Higher-order functions and lazy evaluation allow new levels of modularity to be attained [Hughes, 1989]; this in turn enables programs to be more easily read and understood. The lack of side-effects make the properties of the program easier to reason about and the similarity to mathematical notation can be considered an advantage to those with this formal knowledge. In addition, programmers do not need to concern themselves with storage management; the program is thus free from memory allocation statements and variable declarations [Jarvis, Glaser and van Eekelen, 1995].

The reader will be familiar with many of these topics, including referential transparency, function application, currying, higher-order functions and lazy evaluation. Detail is paid however to abstract types, descriptions of which are used in the discussion of the LOLITA system which is found in the next section.

Abstract types

When using the mechanism of type definitions to introduce a new type, we are in effect naming its values. With the exception of functions, each value of a type is described by a unique expression in terms of constructors. Using definitions by pattern matching as a basis, these expressions can be generated, modified and inspected in various ways. It follows that there is no need to name the operators associated with the type. Types in which the values are prescribed, but the operations are not, are called *concrete* types.

Abstract types operate in the reverse—an abstract type is defined not by naming its values, but by naming its operations. How values are represented is therefore less important than which operations are provided for manipulating them. The meaning of each operation has to be described either by algebraic specification, stating the relationship between the operations as a set of algebraic laws, or by models, describing each operation in terms of the most abstract representation possible[3].

In order to implement an abstract type, the programmer must provide a representation of its values, define the operations of the type in terms of this representation and show that its implemented operations satisfy the prescribed relationships. Apart from these obligations, the programmer is free to choose between different representations on the grounds of efficiency or simplicity.

Important to the design of large programs is the concept of *abstraction barriers* [Bird and Wadler, 1988], the mechanism of hiding the implementation of an abstract type so that the reference to the concrete representation is not permitted elsewhere in the program. In particular, this approach allows the representation to be changed without affecting the validity of the rest of the program. Programming of the system can in effect take place entirely at one of the predefined abstract levels and the maintenance of individual modules can be structured in terms of the abstract operations and types. This is particularly important when the semantics are intuitive from the operations on the type itself; these can be understood without having to understand the mechanics of the underlying implementation. The abstraction may hide a particularly complex implementation, but despite these underlying complexities the model that it produces is an independent language of operations which is convenient and easy to understand. An abstract language may for example be developed for a simple logical language, containing functions for the binary operators *and*, *or* and *not*. This simple language can be understood and used by a variety of programmers, undaunted by the complex model of semi-conductor behaviour which has been programmed below.

[3]These descriptions are taken from [Bird and Wadler, 1988].

Abstract types permit the specification of the data type together with operations which can be performed upon it. In effect this allows a separate sub-language to be designed (since we can only use the provided functions to manipulate the structure) to tackle the given problem. In an imperative language, such sub-languages would normally be implemented with a separate program to parse this language and transform it into code which could then be handled with an imperative language compiler (e.g. the UNIX tool YACC [Johnson, 1975]). The difficulty with this approach is that the tool is difficult to implement in the first place and is subsequently inflexible. It cannot easily be changed to accommodate new or old features. The syntax of the sub-language is different for each tool.

In this thesis the term *domain-specific sub-language* is used to refer to such a programming style. The term *sub-language* is used in preference to the term *language*; although it may seem that new language constructs are being defined, they are still part of Haskell rather than being entirely new languages, and the Haskell syntax still applies. However, if well defined, a domain-specific sub-language can appear to the programmer to be a new language specially tailored to a certain situation.

There are two approaches to designing such sub-languages. The first approach involves identifying commonly used patterns in existing code and creating new sub-language constructs to capture these. The second approach, used in the LOLITA system, involves designing a sub-language best suited to the task and then trying to fit the necessary features into Haskell. Examples of this approach are explored in relation to LOLITA in section 3.4.1.

3.3 The LOLITA NLP System

3.3.1 Natural Language Engineering

The LOLITA system is a Natural Language Processing (NLP) application built within the domain of Artificial Intelligence, or more specifically within the domain

of Natural Language Engineering (NLE). The field of Natural Language Engineering is composed of a number of interconnecting disciplines. It is an engineering activity and is thus pragmatic by nature, though its scientific and technical background is based on Descriptive and Computational Linguistics, Lexicology and Terminology, Formal Languages, Computer Science, Software Engineering and other relevant subject areas.

The engineering of Artificial Intelligence systems, particularly on a large scale, may differ from the construction of large systems built in an alternative domain. There are a number of criteria which are considered in the construction of these systems. Garigliano [Garigliano and Tate, 1995] describes the aims of Natural Language Engineering by emphasising that the systems which this method produces will have, amongst others, the following attributes:-

- **Objectives**: Much of the work produced in the domain of computational linguistics has followed a more traditional approach of modeling small-scale solutions using computer systems. Though these models may be sizable, the solutions which they present may only be small fragments of larger linguistic theories. They may therefore demonstrate a small practical fragment of some idea, though offer no wider solution to the field at large. Natural Language Engineering aims to produce solutions which deal with natural language as a whole. It should also produce solutions which offer a number of pragmatic possibilities (these are expanded upon in the following points), and may lead to systems which are forseeably useful to the general public.

- **Usability**: The usability of a system is characterised by the number of people who are able to successfully use the system. Each of these users will aim to use the system to benefit them in some way, whether it be to reduce work or to extend results and ideas which they may have. These will be the users' *needs*, of which the system will offer a proper subset of solutions. Usability will also include the costs incurred in using the product, this may not simply be in terms of raw monetary value, but also include the time spent installing and learning how to use the new system.

- **Resources**: This is usually considered in terms of what the user is prepared to pay for the system, including hardware, software and data costs. It also includes the human costs such as maintenance and learning mentioned above. Such a figure would normally be calculated through a form of cost-benefit analysis.

- **Scale**: The size of a system will have an influence on all the other parameters mentioned. The scale of a system itself may be divided into a number of sub-parts. For example a Natural Language Engineered system may be built with a need for a wide grammar coverage, but without suitably robust handling of ungrammatical input; such a system would typically be over-engineered. In the same way selecting a small-scale, basic-domain model will require, for most applications, that the data be taken from somewhere else, such as corpus statistics or larger knowledge bases for example. Increasing the size of a system will generally increase the effectiveness of the task. Although the complexity of the solution will remain the same, the time in which the algorithm is executed will grow with the addition of further data and more complex programmed solutions. This increase in the overheads required may finally affect the validity of the solution.

- **Flexibility** Constructing a system for a single domain, to perform a task which is unlikely to change, may be done with little regard to the system's flexibility (reuse, maintenance etc.). In practice however this is rarely the case. Small systems may be constructed in this way, since the savings made by not making the system flexible out-weigh the costs of constructing a new similar system; in the design of larger systems more attention must be paid to producing methods which will ensure the reuse or modification of the system at a later date. This principle applies as much to the core of a system as it does to the components.

- **Implementation**: There is a widespread view that there are programming languages for artificial intelligence systems, just as there are programming languages for business systems and data bases etc. Often such a claim is

based on the experience of programmers within that domain, and the relative expertise that they possess. For Natural Language Processing, as for any other software engineering discipline, this is not true. The choice of the implementation language is based on a number of parameters including the number of developers involved, the languages already known, the data sets used, the architecture on which the system is executed, the interfaces and support tools available and the life-cycle models used. These metrics allow the system designer to select one or more programming languages for the implementation of the system. A solution may be prototyped and later developed in a more portable language.

- **Efficiency**: This aspect has deliberately been left until last, as the consideration of efficiency in Natural Language Engineering is somewhat different from the more traditional views of efficiency expressed in chapter 2. The theoretical analysis of complexity is paramount in a properly engineered natural language system. It is necessary to understand the complexity of the algorithms employed, as an oversight in this part may only manifest itself when the system has been constructed and the operational behaviour proves to be far from what was initially expected. In such systems however, there is a slight anomaly with traditional methods of theoretical complexity analysis. Complexity analysis is in most calculations worst-case analysis; average-case analysis is usually too complex to calculate. Worst-case analysis may identify rare scenarios which would be unlikely to occur during normal operation of the system; this type of analysis can therefore be misleading if a large percentage of the system time is spent in a 'better than worst' case situation. Even when the worst case is the normal one, complexity concerns the system behaviour as the input grows towards infinity. Most input into a natural language system will have its size limited, producing with it a constant which can be used in the estimation of the complexity behaviour of the system. This constant is irrelevant as far as the order of complexity is concerned, yet remains paramount for more practical complexity analysis. For example when considering the processing of a string of words, an algorithm of high complex-

ity (e.g. exponential) may perform better than one of low complexity (e.g. polynomial) when the number of words is bound by an upper limit. Practical complexity analysis therefore has a primary role in natural language systems. Practical complexity analysis may use tools such as parse-graph displaying mechanisms and programming tools such as profilers. The development and maintenance of such systems can be monitored and influenced by the results from these tools (see chapter 4) and they are regarded as an integral part of the development process.

Large systems differ in their perspective as well as in their programming. The LOLITA system at Durham follows the principles of Natural Language Engineering discussed above.

3.3.2 LOLITA

The LOLITA (Large-scale Object-based Linguistic Interactor Translator and Analyser) system is a state of the art natural language processing system, able to grammatically parse, semantically and pragmatically analyse, reason about and answer queries on normal complex texts, such as articles from the financial pages of quality newspapers [Garigliano and Long, 1994].

The development of the LOLITA system began in 1986 when the language Miranda[4] was used. In 1993 the system was entirely converted to Haskell [Hazan, Jarvis, Morgan and Garigliano, 1993]. The system is now used in a number of different research projects in both academia and industry. In June 1993 the LOLITA system was demonstrated to the Royal Society in London.

LOLITA is an example of a large system which has been developed in a pure functional language because it was felt that this was the most suitable type of language to use. The functional approach was chosen originally because of the developers' experience of logical techniques, and the logical consequences of functional languages. Many other large functional systems (discussed at the beginning

[4]Miranda is a trademark of Research Software Ltd.

of this chapter), have been built by functional programming experts; in contrast LOLITA developers often come to the system with little or no functional programming experience, or indeed with no programming experience at all. This differs from others' experiences in developing large functional systems, whose motivation for doing so has often been an interest in functional programming theory.

Expertise gained in the use of functional programming in the LOLITA development team has largely been gained by hands-on development of the system. Many of the techniques documented have been developed out of necessity, rather than being driven by functional programming research. The way in which the non-expert utilises the features of a pure functional language is of interest, as it provides an insight into how large systems would be developed in a more general, possibly non-expert, environment.

The LOLITA system source code is continually being changed. Many of the people currently involved in writing new pieces of code and changing existing code are new to the system. The field in which LOLITA lies means that many of the people working on the system are from non-Computer Science backgrounds; researchers in the group include linguists, mathematicians, computer scientists and economists. As expected the programming experience varies greatly from people who have no programming experience, to those who have experience of imperative languages, and to those who are proficient functional programmers.

In spite of these apparent obstacles, alterations to the system have been accomplished with surprising ease and very little disruption to other parts of the system and other LOLITA developers' work. Novice functional programmers have relatively effortlessly incorporated their work into the system. People who previously had little or no idea of how the LOLITA system worked have been able to commence with their modifications in a matter of days. Of course the situation is not entirely perfect, but given such an unstructured decentralised development model, the evolution of the LOLITA system has been able to proceed remarkably smoothly.

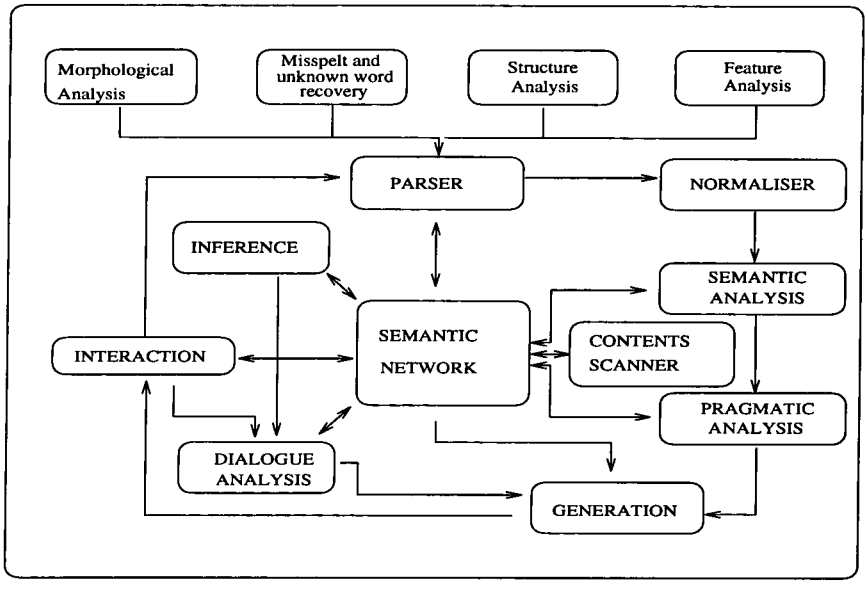


Figure 3.2: Structure of the LOLITA system.

In the following sections the LOLITA system is introduced, with a brief description of some of its applications and an outline of the system design. Lazy evaluation has been highly influential in determining the overall structuring of the system; this is explained in Section 3.4.5. The technique of creating *domain-specific sub-languages*, a feature central to the programming and development of the LOLITA system, is introduced in Section 3.4.1; the facilities provided by Haskell which contribute to the ease of implementing and using such sublanguages are discussed. In Section 3.4.7 the handling of state in LOLITA and the contribution of purity to the ease of implementing multiple semantic networks are examined. Debugging and profiling tools and techniques used with the LOLITA system are discussed in Section 3.5. Finally some conclusions drawn from the use of Haskell to code LOLITA are presented in Section 3.6.

3.3.3 System construction

Many natural language processing systems have been built to solve specific problems. These systems are restricted, either by a particular task which they perform, or by the domain in which they work. The aim of LOLITA is to produce a general, domain-independent knowledge representation and reasoning system.

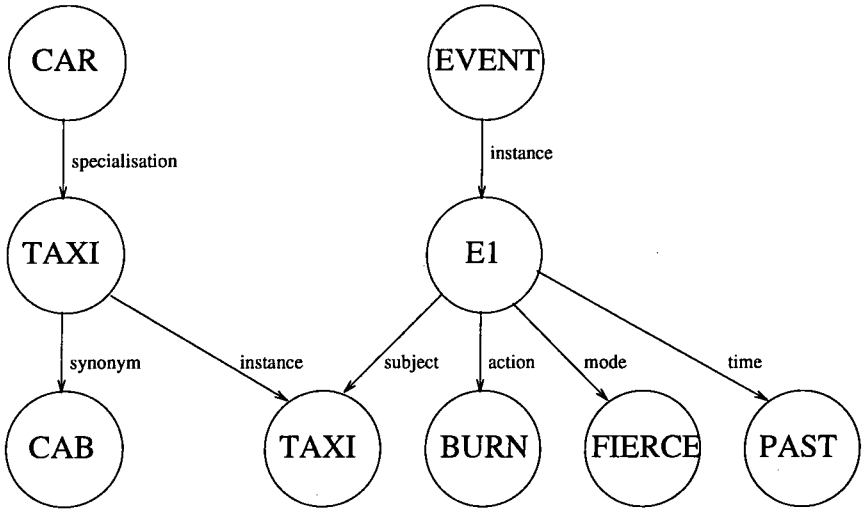


Figure 3.3: A portion of the semantic network.

The overall structure of the LOLITA system can be seen in Figure 3.2. The core of the system is a general framework which is used to map from text to meaning and meaning to text. The main data structure used to represent this meaning is a *semantic network* [Shastri, 1988][Garigliano and Long, 1994]. This semantic network, *SemNet*, is a structure which stores information independently of any natural language representation. It is inheritance-based and holds world information and data, as well as some linguistic information.

Figure 3.3 shows a simplified portion of the semantic network representing the event 'the taxi burned fiercely'. The transformation from text to meaning is carried out by the parser, the normaliser, the semantic analyser and the pragmatic analyser. Each language understood by LOLITA (the system currently has dictionaries for English, Italian, Spanish, Chinese and French) requires the construction of a syntactic parser [Ellis, Garigliano and Morgan, 1993] to map from text to semantic net. The English parser, for example, contains over 1,500 grammatical rules, written using the method described in section 3.4.1. After syntactic parsing the parse tree is normalised—equivalent parse trees are mapped to a unique normal form. The semantic analyser then transforms the parse tree provided by the normaliser into a fragment of semantic net, and matches the nodes it creates with those which currently exist. The pragmatic analyser then ensures that the meaning produced by the semantics is consistent with LOLITA's knowledge of the real world.

3.3.4 Applications of LOLITA

The LOLITA system has been designed to maximise flexibility. The core system, described above, comprises the majority of the code—application-specific code makes up only 0.5% of the total code in the contents scanning example described below. This independence of the core system from a specific domain makes the development of a new application relatively straightforward [Hazan, Jarvis, Morgan and Garigliano, 1993]. Applications which have been developed using the LOLITA system include:

Contents scanning

Contents scanning involves text being examined and information contained in the text being used to fill in an outline template. Contents scanning [Garigliano, Morgan and Smith, 1993] is one of the standard tests of the abilities of a natural language processing system. The most widely known and acknowledged of such tests is that of the Message Understanding Conference (MUC), run by DARPA in the United States[5] [Morgan and Garigliano et al, 1995]. An example of contents scanning, as performed by LOLITA, is given in Figure 3.4.

In the LOLITA contents scanner, the input text is parsed and semantically analysed in order to build a representation in the semantic network. A domain-dependent module then searches the network for information relevant to each of the slots. This information, in the form of semantic-network nodes, is then passed to the realizer which produces the output.

Chinese tutoring

LOLITA has been used as the core engine for a system to aid the teaching of Chinese to English-speaking students [Wang and Garigliano, 1992]. One of the main problems encountered in the learning of foreign languages is the influence of

[5]LOLITA entered the MUC-IV competition.

A car bomb exploded outside the Cabinet Office in Whitehall last night, 100 yards from 10 Downing Street. Nobody was injured in the explosion which happened just after 9 am on the corner of Downing Street and Whitehall. Police evacuated the area. First reports suggested that the bomb went off in a black taxi after the driver had been forced to drive to Whitehall. The taxi was later reported to be burning fiercely.
(THE DAILY TELEGRAPH 31/10/92)

```
Template: Incident
Incident: A bomb explosion.
Where   : On the corner of Downing Street and Whitehall.
          Outside Cabinet Office and outside 10 Downing Street.
          In a black taxi.
When    : 9pm.
          Past.
          Night.
          When a forceful person forced a driver to drive a
          black taxi to Whitehall.
Responsible:
Target: Cabinet Office.
Damage: Human: Nobody.
        Thing: A black taxi.
Source: telegraph
Source_date: 31 October 1992
Certainty: Facts.
Relevant Information
         Police evacuated 10 Downing Street.
```

Figure 3.4: Example of the contents scanning task.

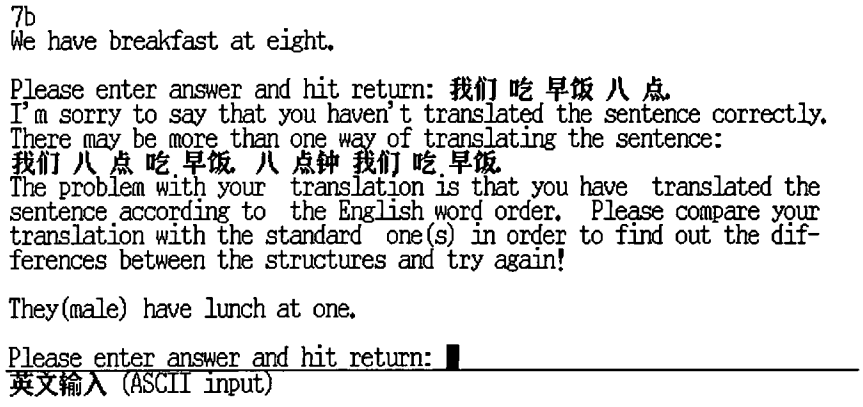


Figure 3.5: An example output from the Chinese Tutor.

the mother tongue, known as *negative transfer* [Selinker, 1969]. This is the use of native language rules or patterns which leads to an error or inappropriate use in the target language.

The Chinese tutor makes use of a technique of *mixed grammar*. When the parse of a Chinese sentence fails, a partial English grammar rule is invoked for the next part of the sentence. A backtracking algorithm is used in order to cope with incorrect parsing.

The mixed grammar of Chinese and English has been modelled in such a way as to allow the parser to locate complicated transfer errors, not only by examining the error itself, but also by checking its links with other constructs in the sentence. Moreover, the grammatical rules in the mixed grammar of Chinese and English can be used to pinpoint arbitrary transfer errors made by students, without predetermining where the errors might occur. The student can then be informed of any errors, and the system can recommend remedial work.

Dialogue analysis and generation

It is possible for a user to carry out a dialogue with LOLITA [Jones and Garigliano, 1993]. This dialogue takes the form of a rich natural language interaction between LOLITA and the user. The dialogue system takes into account factors which influence everyday dialogue between humans, such as the emotional state of the

```
YOU:    I like you.
LOLITA: I know.
YOU:    You are beautiful.
LOLITA: I am glad because you think I am beautiful.
YOU:    Do you like me?
LOLITA: Yes, I like you.
YOU:    Why do you like me?
LOLITA: I do not know why I like you.
YOU:    I am tired.
LOLITA: I understand.
YOU:    Are you tired?
LOLITA: No.
YOU:    Tell me if you believe that I am tired.
LOLITA: Yes, you are tired.
```

Figure 3.6: Example of a dialogue with LOLITA.

participants, their motives and the relationship of one participant to the other.

The LOLITA system utilises a Dialogue Structure Model [Garigliano and Jones, 1992], which is a template-like schema containing the basic information for the dialogue structure pertaining to a given situation, for example having a chat or attending a job interview. This information is then used to structure the responses that LOLITA will give, so that they are applicable to the current situation. The parameters of the dialogue are optimised to an intended script using an evolutionary programming algorithm [Nettleton and Garigliano, 1994]—this exemplifies the eclecticism of the methods used within the system. A sample dialogue between LOLITA and a user is shown in Figure 3.6.

Other applications

- Query application — This allows the user to have a 'question and answer' session with LOLITA. It is possible to 'tell' LOLITA information and then ask questions about this information or other information already held in the semantic net. This differs from the dialogue feature in that the style of the interaction is plain—no account is taken of the context of the protagonists involved in the exchange.

- Machine translation — Although this was not part of the original system, it has been possible to add the functionality of a prototype translator from Italian to English to the system with relatively little effort. Sentences can be parsed according to the rules of Italian grammar and the information added to the semantic network. LOLITA's generation system [Smith, 1995], which takes sections of the semantic network and produces the English text, can then produce the English from this network.

- Story application — This is intended for use by physically handicapped people. The user is presented with a list of events, in the form of short sentences. The user is able to select events for inclusion in his story, these events are then processed by LOLITA , which will generate a paragraph based upon them.

3.4 Aspects of Large-Scale FP

The extension of the core system for each application is possible due to the different programming practices employed. Developers of the individual applications may not know how other parts of the system work and they rely on a set of interfaces which supply the necessary functions.

It would be expensive in terms of time and effort to teach each of the new members of the group a comprehensive selection of functional programming practices, perhaps including primitive data type mechanisms, defining algebraic types, monadic definitions and state threading. However, it is clearly necessary to teach the programmer some aspects of functional programming; for example function construction and application, and primitive data types might be two useful mechanisms. These can then be combined with the programmer's knowledge of the abstract data types he is employing. Additional knowledge of more advanced functional programming techniques such as defining algebraic types and monadic definitions need not be taught.

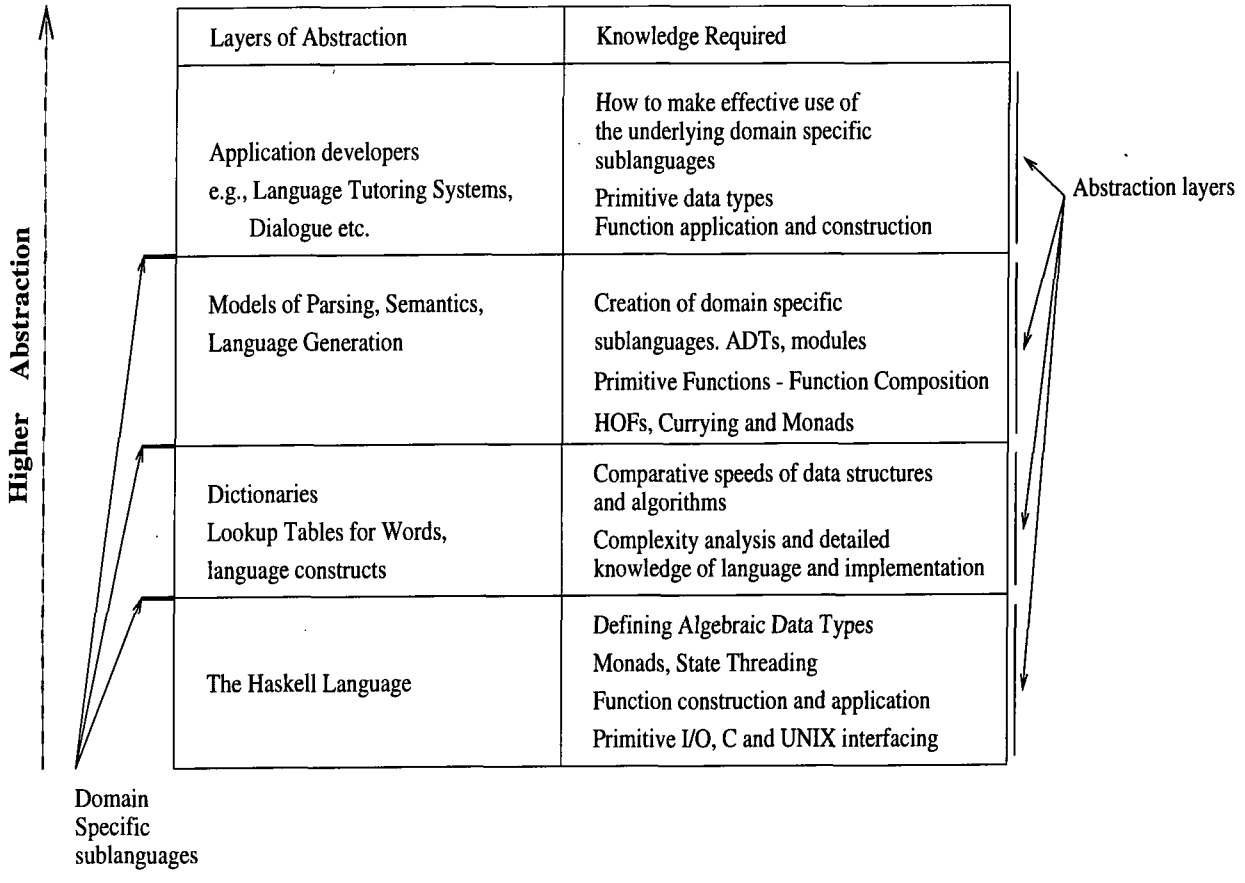


Figure 3.7: A diagram showing the various levels of abstraction that exist. The column on the left shows the layers of abstraction (based on the Haskell language) at which applications are developed. People developing at lower layers support those at higher layers by providing tools and/or creating a level of program abstraction. The second column shows the knowledge that those working at each level require. Each level of abstraction can be bridged by the use of a Domain-Specific Sublanguage. The real power of Functional Languages lies in the way in which the boundaries between these levels of abstraction can be drawn [Jarvis, Poria and Morgan, 1995].

```
> reported_typesentence :: Parser
> reported_typesentence
>   = prephrases
>     +++
>     typesentence &? exclamation_mark    >> exclamativeN
>     +++
>     typesentence & question_mark        >> questionN
```

Figure 3.8: An example grammar rule.

This approach of instructing the developers at various abstract levels aims to minimise the amount of teaching which needs to be done. With developers working at different levels of abstraction a hierarchy of programmers is created. This hierarchy has naturally developed into the levels seen in Figure 3.7. Functional programming techniques allow these levels of abstraction to be created with a number of different features which are examined in the following sections.

3.4.1 Abstract types for Domain-Specific Sub-languages

One technique which is used throughout the system is the use of abstract data types to create *domain-specific sub-languages* appropriate to specific natural language tasks (introduced in section 3.2.2 and expanded upon here). This is particularly appropriate for parts of the system which rely on large numbers of rules. For example, Figure 3.8 shows one of approximately 1,500 grammar rules currently contained in the system.

Although this rule is actually a piece of Haskell code defining a value of type `Parser`, it directly corresponds to the standard formulations of grammatical rules. In particular, when the semantics of the various operators are explained, the rule has clear declarative semantics. The definition given in Figure 3.8 states that a `reported_typesentence` is either a prepositional phrase, a `typesentence` followed by an optional exclamation mark symbol or a `typesentence` followed by a question mark. It also states that if the second form is found, a parse tree should be built

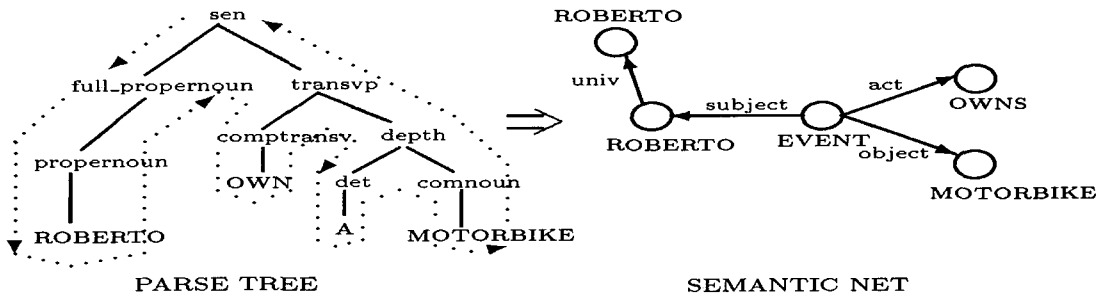


Figure 3.9: A fragment of the semantic net.

with the label `exclamativeN` and if the third form is found, the parse tree should be labelled as `questionN`.

It can be seen that the identifier `reported_typesentence` and the identifier `exclamation_mark` correspond to non-terminal and terminal symbols respectively; the `+++` operator is used to indicate alternatives; `&` indicates sequencing; `&?` indicates sequencing with an optional right part and the `>>` operator gives the label to be used in the resulting parse tree.

As well as the domain-specific sub-language for the parser, other domain-specific sub-languages have been designed for different tasks in the LOLITA system, most notably for semantic analysis and natural language generation. The following section briefly describes the semantic parser and gives an overview of the domain-specific sub-language used to implement it.

3.4.2 The semantic parser

The semantic parser is a central feature of the LOLITA system. The input to the semantic parser is a syntactic parse tree and its output is the corresponding semantic-net structure. The semantic parsing stage therefore involves the transformation of information from the parse-tree structure to the semantic-net data type. An example fragment of the semantic net, corresponding to the sentence

"Roberto owns a motorbike", is shown in Figure 3.9. Each node in the parse tree is labelled with its grammatical construct. For instance the root node of the parse tree is labelled with `sen`, representing the complete sentence structure. For each of these labels, there is a corresponding semantic rule which transforms the parse-tree structure into the semantic-network structure.

One type of semantic rule is used for leaves in the parse tree and another is used for branches. The parse-tree leaf rules are represented by the abstract type `LeafRule`; a function `metaLeaf` takes a parse-tree leaf label and returns the corresponding leaf rule. Similarly, a function `metaBranch` takes a parse-tree branch label and returns the corresponding branch rule, of type `branchRule`:

```
> metaLeaf :: ParseTreeLabel -> LeafRule
> metaBranch :: ParseTreeLabel -> BranchRule
```

The semantic representation of a node in the parse tree is thus determined according to its label and the semantics of the subtrees below it. Taking the `transvp` node of Figure 3.9 as an example, the left subtree produces the concept of ownership and the right subtree produces the concept of a particular (but unspecified) motorbike. Because these are linked together by a branch labelled `transvp`, the ownership must be an action and the motorbike must be an object. This rule is specified in the semantic analysis domain-specific sub-language as follows:

```
> metaBranch "transvp"
>   = labelBoth Act Obj
```

Although the rule for `transvp` can define the semantic representation of a node entirely in terms of the semantic representation of its subtrees, other rules must take into account contextual information available at that point, such as the set of referents, nodes which may be referred to in later pieces of text by pronouns (e.g. 'he' or 'it'). It must be possible to mark points in the parse tree which may correspond to new nodes in the semantic network as well as points which may be

referred to later in the text, additional operators and functions are provided for this purpose.

Another feature which is necessary for semantic analysis is the ability to combine two or more rules. The operator `compose` does this, applying the first rule to the result obtained from the application of the second; it is the semantic rule equivalent of function composition. The following example creates a branch rule for proper nouns by combining the three smaller rules `labelLeft`, `newnode` and `addref`:

```
> metaBranch "full_propernoun"
>  = addref          .
>    newNode object  .
>    labelLeft Univ
```

This rule specifies the semantics for `full_propernoun` as a unique new object node related to the semantics of the left subtree by a universal link. This branch contains no right subtree, as can be seen in the example of Figure 3.9, hence the `labelLeft` rule is used. The `newNode` rule is used to distinguish between a specific instance of the object (ROBERTO in the example) and the concept of the universal set of ROBERTOs. The `addref` rule ensures that this new node is available as a referent; this would be used in the sentence "Roberto owns a motorbike and *he* cleans it almost every day" to enable the semantic analysis to work out that *he* refers to Roberto.

3.4.3 Semantic analysis implementation

Semantic analysis specified in the manner described above is implemented as a folding operation on the parse tree. Since the sub-language needs to support context and allow for the construction and joining together of pieces of semantic network, this tree folding operation must be performed in the presence of a state which is threaded from left to right in the parse tree. The rules themselves are represented as functions which modify this state as well as producing the appropriate semantic result to be made available to the level above.

The major benefit of the use of this abstract data type is that it protects the person writing the semantics from the need to plumb the state explicitly between different semantic rules. It has also allowed for the addition of new features, such as semantic alternatives and rejection of parse trees which are semantically incorrect. These new additions only affect the implementation of the abstract data type and the rules which need to take advantage of the new facilities.

3.4.4 Analysis of the domain-specific sub-language approach

It can be seen from the above examples that sets of rules such as grammatical and semantic rules are defined at an abstract level, rather than at the level of primitive data structures. In LOLITA each domain-specific sub-language is implemented using layers of abstract data types, with inner layers describing lower-level operations and the outermost layer being the domain-specific sub-language itself. An alternative to the approach used in the LOLITA system would be to define a completely new sub-language in which the rules could be written and then write appropriate tools to deal with the sub-language, either by generating code for it or by performing some form of analysis and interpretation, using tools such as YACC.

This alternative method of implementation would give two significant advantages over the current approach. It would allow the designers of the rule sets complete notational freedom and would also allow the rules to be preprocessed off-line. However, the current approach does have significant advantages:

- **Initial Cost** — Although the cost of implementing the semantics of the sub-language under either approach is similar, as both could be done in Haskell, the cost of dealing with the syntax would be much greater if a completely separate sub-language were used, as it would involve writing a new parser, at the very least.

- **Flexibility** — A domain-specific sub-language may be changed simply by adding new functions or altering existing ones. This is something which has occurred frequently during the development of LOLITA.

- **Power** — Domain-specific sub-languages are extremely useful when an appropriate set of constructs can be devised to cover all of the rules without becoming too complex. However, in a substantial rule set there are often certain rules which require special treatment. With the current approach it is easy to revert to the full power of Haskell, albeit with the loss of some abstraction at isolated points. In the external sub-language approach this would be far more difficult to achieve, as it would require either the implementation of ad hoc primitives or some facility to interface with a more powerful sub-language.

- **Scale** — The overheads in setting up new language tools mean that the external approach is only feasible for substantial rule sets. The internal approach has very small overheads and is thus applicable to much smaller rule sets.

It would certainly be possible to use this domain-specific sub-language approach in other languages. Most recent languages provide facilities for the creation of abstract data types; these are essential to the use of domain-specific sub-languages as they prevent the user of the sub-language from accessing the implementation of the types used in the sub-language directly. The ability to define operators with specified precedence and associativity is also provided in other languages.

However, two features of Haskell make it particularly suitable for creating and using domain-specific sub-languages. The first feature is lazy evaluation. The use of lazy evaluation in domain-specific sub-languages is discussed in the next section. The other important feature, the use of higher-order functions to provide a further level of abstraction, is then discussed.

3.4.5 Lazy Evaluation

An important application of lazy data structures in the LOLITA system is the handling of the parse forest from the LOLITA parsing stage. The LOLITA system uses the Tomita parser [Tomita, 1986] to produce a set of parse trees, known as a parse forest, for a particular sentence or group of sentences. A post-parse stage

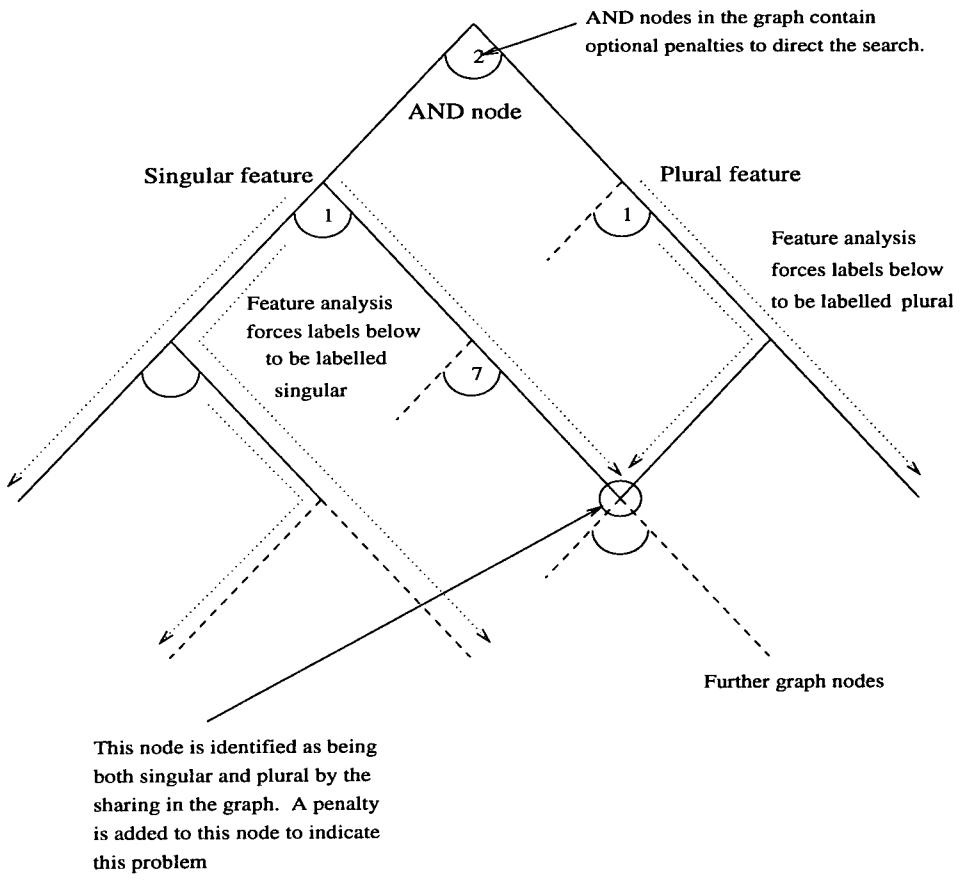


Figure 3.10: Feature analysis during the post-parse stage of analysis.

builds a collection of parse trees from this parse forest, performing feature analysis on the parse graphs at the same time.

Feature analysis considers parts of the parse graph and forces other parts of the graph to adopt the same syntactic feature. For example if a node in the parse graph has the feature *singular* attached to it, then feature analysis forces other parts of the graph around this node to have singular features too. The nature of the parse graph structure means that it may be possible that the node in the graph is also referred to by another part of the graph, which may itself have a *plural* feature attached. This makes the analysis considerably more complicated, see Figure 3.10.

Feature analysis occurs from the top of the graph and forces features down the graph depending on what feature analysis is needed at that point in the graph. If the features do not match at a single point in the graph then penalties are added to indicate the likelihood of that feature analysis being correct or incorrect.

Optional penalties are also included at the *AND* nodes in the graph. These penalties are used to control the search.

During the post-parse stage a table data structure is built which will have reference to any node in the graph. It is possible to look up nodes in the graph under any particular combination of forced features. There may therefore be a version of a graph node where the feature *singular* is forced and also a version of the same node where the feature *plural* is forced. Including all the possibilities of nodes creates a potentially huge table.

To reduce the amount of computation necessary to construct parse trees from the table, the table is defined as a cyclic structure. Computing the node at one point in the graph is aided by using the look-up table to reference the nodes lower down in the graph. Using this look-up table there is no need to build the sub-graphs from scratch; if the node has already been visited by some other route, then the computation of its value is no longer necessary.

This look-up table is defined using lazy evaluation. The potentially large table is only built where it has been visited by the search; this is determined by the penalty scores which control the route of the search. Using lazy evaluation also allows parse trees which have already been constructed to be included in the new parse trees built higher up in the graph. Sharing can take place so parse trees higher in the graph may refer to those created below. Lazy evaluation allows such a process to take place.

This process is defined in terms of a function, `x`, which creates the look-up table, `mkLookupTable`, which in turn takes the same function `x` as its argument. This cyclic definition is necessary because as a particular node in the look-up table is being built, it might also be necessary to access nodes from the look-up table which have already been built[6].

```
x = mkLookupTable x
```

[6]This piece of code is simplified for the sake of the example; the actual code is recorded over the page.

```
> mkFeatureForests :: ParseForest -> FeatureForests
> mkFeatureForests pf
>  = ff
>    where
>       ff = FeatureForests (length ffs)     $
>            [listToFM $ map (mk_entry (pf,ff)) fs | fs <-gramFeatSets]
```

Figure 3.11: The `mkFeatureForests` function takes a parse forest as an argument, this is the result of the LOLITA Tomita parser. It builds as its result a feature forest with the set of features (`gramFeatSets`) supplied. The function has a lazy cyclic structure which can be identified by the reference to the feature forest `ff` in the definition of the feature forest itself.

```
> data GenVal a b = Val (a -> b)

> ifGen :: GenVal a Bool  -> Generator a
>                 -> Generator a -> Generator a

> ifGen (Val cond) thenPart elsePart
>    = Generator (\w -> (f w) w)
>      where
>         f x | cond x    = thenPart
>         f x | otherwise = elsePart
```

Figure 3.12: The generator conditional function.

Lazy evaluation firstly makes such a definition feasible; in a strict language it would not be possible to pass the look-up table `x` as an argument until it had been built. Secondly, the lazy definition means that only those parts of the look-up table which are needed are built. This is very important since the table is potentially very large. Lazy evaluation makes cyclic definitions such as these possible. The actual code for this definition is found in Figure 3.11.

This is not an isolated example within the LOLITA system. The abstract type used in the generator part of LOLITA also relies upon the laziness of function application. The `Generator` abstract type provides a conditional construct called `ifGen`. This construct evaluates either its second or third argument depending upon the value of its first (Figure 3.12).

3.4.6 Higher-order functions and parameter hiding

When using domain-specific sub-languages, a programmer will often apply a function to a value without realizing that these values are themselves functions.

The LOLITA generator provides an interesting example. The generator takes information from the semantic net of the LOLITA system and combines this with planning instructions in order to generate text. A simplified version of some of the generator code is shown in Figure 3.13.

The code fragment shows a number of domain-specific functions which programmers of the generator are able to use in the creation of their code. These functions, including `if_` and `` `or_else` `` (1,2), are not Haskell constructs but functions in their own right.

The function `if_gen` (3) queries the hidden parameter passed by the Generator type; the `say_` functions (4) generate phrases for each role event in the semantic net. Each of these functions returns a `Generator` function and the function `before_` (5) is used to compose these generator functions together. Despite being complex higher-order functions, they are used as basic constructs to the generator language.

The result of using this form of representation is to make functions written using these abstract types easier to write and clearer to read. Currying allows the input parameter to be omitted from function definitions; this can be seen in the generator examples which are written as rules rather than functions. The programmer is thus writing at a functional level without realizing it; each function appears to be a rule or a language construct rather than a function.

3.4.7 Purity and referential transparency

In this section some of the more noteworthy consequences of choosing a pure functional language to code the LOLITA system are discussed. Purity may sound intuitively like a 'good' property for a programming language to possess, but to many people this appears not to be true, as Hughes points out in [Hughes, 1989].

```
say_meaning :: GenVals -> Meaning -> Generator
say_meaning gv n
  = if_ (is_event_m n)                                    (1)
        (if_ (forced_closed_event_gv gv)
              say_event_as_noun gv n
         `or_else`                                        (2)
              say_event gv n
         )
    `or_else`
        say_entity gv n
        .
        .
        .
        etc

say_event :: GenVals -> Meaning -> Generator
say_event gv e
  = if_gen (is_style active)                              (3)
       say_active_event gv e
    `or_else`
    if_gen (is_style passive)
        say_active_event gv e                             (4)
        say_entity gv n
        .
        .
        .
        etc

say_active_event :: GenVals -> Meaning -> Generator
say_active_event gv e
  =  say_subject gv e
    `before`                                              (5)
     say_action gv e
    `before`
     say_object gv e
        say_entity gv n
        .
        .
        .
        etc
```

Figure 3.13: Simplified portion of the generator code.

Indeed in LOLITA it has proved to be necessary to incorporate certain impure features, for example for retrieving the current system time and date and for debugging (Section 8.2)[7]. However, apart from impurities such as these, LOLITA is coded entirely in a pure fashion. Many of the benefits of purity are widely known, ease of proving the properties of programs and implementing correctness-preserving transformation being two examples. However the example which follows is one in which the use of a pure functional language led to a surprising advantage over the use of an imperative language.

Multiple semantic nets

The operation of the semantic net was initially conceived as an abstract state machine and would have been implemented as such had an imperative language been used—a single copy of the semantic net would have been operated upon and altered by a set of procedures. Naturally, in a language possessing the property of referential transparency, this implementation is not possible, as it relies on the use of side-effects to alter the semantic net.

The technique used to implement the semantic net in LOLITA involves passing the state explicitly as a parameter from one function to the next. This was initially seen as a serious disadvantage of using a functional language. However, although the state is a very large data structure, there is very little loss of efficiency in doing this, as the language implementation means that only a pointer is passed each time. In fact, the outcome of using this approach has been that altering LOLITA to use multiple, different copies of the semantic net has been rendered very straightforward.

The semantic analysis stage involves adding information to the semantic net and altering existing information. There are often a number of alternative analyses which must be explored and hence a different semantic net is required for each. This was not a feature that had been thought of when LOLITA was first

[7]Note that it is possible to do this in a pure manner using monads to 'plumb' this behaviour; however it would require a substantial amount of effort in terms of re-writing the code.

implemented; the semantic-net implementation had not been designed with the possibility of this additional functionality taken into consideration. If LOLITA had been implemented in an imperative language, with the semantic net as an abstract state machine, much alteration would have been required in order to introduce this new feature: each function which used the semantic-net data structure would have needed to keep track of which semantic net was being used at the time and a large amount of extra storage would have been needed for the new copies of the semantic net. In the actual implementation, however, passing a different copy of the semantic net to a function does not affect any other parts of the program.

To include this additional requirement in the LOLITA system, it has not been necessary to alter the implementation of the semantic net in any way. So easy is this feature to incorporate that it is not even necessary to know whereabouts in the program multiple copies of the semantic net are being used; it is a facility which is taken for granted. Without the explicit references to the semantic net, dealing with multiple versions would have been much more difficult, especially in the presence of lazy evaluation, which would have prevented any clear notion of a 'current' semantic network.

Impact on programming style

An obvious consequence of programming in a pure functional language is that those developers who are used to programming in an imperative language must learn to change their programming style. Although many of the LOLITA developers initially found it difficult to adjust, it would appear that this was soon overcome. Indeed, it is interesting to note that once they became used to programming in a lazy functional language, many found that the new style of programming influenced their writing of code in imperative languages—an increased use of recursion is one effect that has been mentioned.

3.5 Programming tools for LOLITA development

Programming tools are an essential part of the imperative programmer's toolkit. Large, real-world systems in particular benefit from mechanisms which allow the developer to *see* what the program is doing at different instances in time; this may indicate why the program is not working, by supplying some debugging information, or why the program is using large amounts of memory or takes an unexpected amount of time to execute, a mechanism demonstrated by profiling tools.

It could be argued that the success of imperative languages is partly due to the ambitious collection of tools, the armory, with which the developer can work. Functional programming languages also benefit from these tools; profiling tools for functional programs were discussed in detail in the previous chapter.

Programming tools provide the LOLITA developers with an important set of means to improve and maintain existing code, and to aid the development of new code. Although the LOLITA development team at Durham have had a certain amount of success with existing programming tools[8], it has been necessary to develop our own more advanced set of tools to meet the needs of the multi-skilled programmers at Durham.

3.5.1 Debugging

Debugging pure lazy functional programs is often regarded as being much more difficult than debugging imperative programs. This is mainly because most of the debugging methods available to programmers in other types of language cannot be used in lazy functional programming.

Because of this, new methods of debugging have had to be devised for use with the LOLITA system. Most of these involve manually altering the source code to provide extra debugging functionality; however, an automated debugging tool has also been developed. A majority of the methods have been developed for use with

[8]The analysis of using profiling tools in the maintenance and development of the LOLITA system is discussed in chapter 4.

specific modules or for debugging specific types of error.

Although Haskell does not enforce it, specifying the type of each function is an important aid to debugging—if the type of the function inferred by the type-checking system is not the same as that specified by the programmer, this is reported as an error, which saves time later on as the programmer does not have to decide whether the error is in the function itself or in the calling function.

It is occasionally stated that the provision of strong typing in pure functional languages means that debugging tools are not required. Although many of the errors are indeed caught by the type-checking system, there will be errors remaining in the logical behaviour of the program which need to be detected and fixed. Experience with Miranda has shown that using an interpreter can be very helpful when isolating errors, enabling the testing of individual functions much more easily than with a compiled program, but this by itself is still not sufficient. In many cases it can be very time-consuming; several different methods have thus been devised to aid the debugging.

Imperative-style traces

The difficulties of tracking the application of grammar rules in the syntactic parser led to the development of a system of *imperative-style traces* to help in debugging the grammar rules. These involve the use of side-effects to print out the name of a grammar rule when it is applied. The `trace` pseudo-function, which takes a grammar rule and a string argument and returns the grammar rule, is used to cause this side-effect. Tracing may be turned off by substituting a dummy definition for the `trace` function.

Although this form of tracing can be useful, it involves the use of side-effects and it can generate far too much information, forcing the programmer to devote a large amount of time to interpreting the debugging output. The program must also be recompiled in order to turn the tracing on or off.

Distinguished Path Debugging Tool

A tool which avoids the problem of too much output has been developed [Hazan and Morgan, 1993]. A class of run-time error which was found to be occurring frequently in the LOLITA system was the *exception error* type. An exception error is one which results in termination of the program and the printing of an error message. Examples of this type of error in Haskell are

```
Fail: head{PreludeList}: head []
```

which results from passing the `head` function an empty list and

```
Fail: (!!){PreludeList}: index too large
```

which occurs when the list indexing operator is passed a subscript which is outside the bounds of the list. These exception errors give the programmer no indication of whereabouts in the program the error occurred. This is a problem as functions such as `head` and the list indexing operator are used many times in many different parts of LOLITA. Previously, this problem had been approached by providing a customised version of each function capable of generating an exception error for each module. This new version of the function would report the name of the module when it generated an exception error. However simply knowing the name of the module in which the exception was generated is not sufficient—the exception error may be in a function which called the exception-generating one, or even some way back in a chain of functions each calling the other with the exception-generating function at the very end of the chain.

The *distinguished path debugging tool* allows the display of these chains of functions, termed *distinguished paths.* The path displayed is the route taken through the dependency graph of the functions in the program. The tool works by transforming each function to take an extra parameter, a representation of the distinguished path, which is built up from one function call to the next. When an exception error is encountered, the value of this extra parameter is displayed. Unlike the previous

method of debugging such errors, which involved altering the source code by hand, the tool works automatically by transforming each source module.

Unfortunately the development of the distinguished path debugging tool has been based on the Miranda system and when the conversion of the LOLITA system to Haskell took place it was updated to the Gofer system. It is not possible to run LOLITA using the Gofer system, as the size of the system and the environmental interaction (see section 3.6) insist that it needs compiler support. The development of debugging tools for the LOLITA system will be revisited in chapter 8 as the creation of a more detailed profiling tool (described in this thesis) has produced the necessary support to re-implement the distinguished path debugging tool on the Glasgow Haskell Compiler.

3.5.2 Profiling

The subject of this thesis is the development of a more advanced profiling tool for use on the LOLITA system. Chapter 4 describes the use of current profiling tools during the development and maintenance of the LOLITA system and the problems which were encountered. The LOLITA development provides a unique example for the testing of such tools and from the experience gained a new profiling tool is proposed, developed and tested; chapters 4, 6 and 7.

3.6 Conclusions

The Dagstuhl workshop on Functional Programming in the Real World identified a number of large-scale systems developed in a functional programming language. Many of these systems were written in the strict functional language ML (or an ML derivative); of the remaining systems, written in a lazy functional programming language, only five were written outside the main functional programming research universities. The LOLITA system at Durham appears to be the largest of these systems.

There are a number of facilities which our own experience of program development in lazy functional languages have identified. There are also a number of possible areas for the enhancement of functional programming languages in general, and Haskell in particular, which have been noted:

- An interpreter system — The conversion from Miranda to Haskell brought a number of problems, one of which was the lack of a fully operational interpreter system for the language. The design of new sections of code and system functionality was clearly aided by the use of the Miranda interpreter; functions at any level in the system could be tested quickly and easily by any member of the development team. Once the designer was satisfied that the functionality was correct the integration proved to be a simple task. Whilst accepting that HBI and HUGS [Jones, 1995] provide a basic interpreter for Haskell and that code can be tested at a very low level, these systems do not fully support the features required for large-scale implementations: HUGS does not support the Haskell module system and HBI does not deal with recursive modules or low-level system support. There would be a number of benefits of a fully supportive interpreter for the Haskell language particularly for applications programmers.

- Abstract form of pattern matching — would add further support to the domain-specific sub-language approach which is paramount in the development of the LOLITA system.

- Preprocessing by compiler — for example grammar transformations, would also add further support for the creation of domain-specific sub-languages without creating the need to write code transformation tools.

- Timeouts and exception handling— are a useful support for systems which are based in the real-time reactive environment. The purity of a functional language must be supported with environmental features offered by other programming paradigms. Without these features functional languages will not support the development of large systems and many programmers will

be forced to integrate with other languages such as C. Development on the LOLITA system has forced the Haskell designers to provide this support which is now found in Haskell 1.3.

- Operating System interface — Interface with machine code and C. The LOLITA development has again provided material to necessitate the integration of these features into the Haskell compilers. The integration of these features has been provided, however no standardisation has yet been proposed.

- Environment support tools — It has been necessary to create our own support environment during the development of the LOLITA system. A system of elaborate Makefiles, RCS archiving, local and global file updates and the checking in and out of working modules has been developed. This support is necessary in the development in a large system with a sizable number of programmers. Haskell offers little in terms of a development support environment; it would be desirable to have standard debugging tools, standard compilation and linking mechanisms (*HBC make* is certainly a step in the right direction), more advanced tracing mechanisms, larger code libraries, larger availability (PCs etc.), a shell environment and function-orientated editor in which to work. This would improve the chances of functional languages becoming more popular.

- Debugging and profiling tools — are necessary on a wide scale if multi-skilled programmers are to be able to develop systems in functional languages. These tools should not be restricted to those programmers with a detailed understanding of the underlying implementation, or those who recognise low-level implementation details. Rather, they should operate at all levels of abstraction and offer support for the successful interpretation of their results.

- Compilation problems — Although extensive improvements are being made to the time in which functional code takes to compile, in the worst case (when the system needs completely re-compiling) the LOLITA system still takes a considerable amount of time in which to compile. There are also a number of

incompatibilities between different implementations of the Haskell compiler[9] which mean that it is not possible to successfully move between versions without careful attention and alterations to the code.

Despite the problems with the current functional program implementations which have been highlighted, the development of the LOLITA system has also identified a number of features of lazy functional programming languages which are particularly good for these types of system.

It is argued that functional programming languages are easier to comprehend than their imperative counterparts. Features such as referential transparency, functional application and currying, abstract data types and higher-order functions make the code more compact and easier to understand. The comprehension of existing code is essential in the maintenance and development of a large system. Purity is a useful aid for the integration of new code and offers programming techniques which could not be easily achieved in an imperative language.

It was noted that functional languages are particularly good for the implementation of domain-specific sub-languages, which in turn play an essential part in the development of the LOLITA system. It is incorrect to generalise this conclusion to the creation of domain-specific sub-languages in the development of all large-scale lazy functional systems. However, conclusions can be drawn which can either be supported or not in the development of similar systems in the future.

The way in which the LOLITA developers use abstract data types offers a powerful and flexible approach which can be used with a fall-back to the standard Haskell system in special circumstances. This has proved successful in an enormous amount of the system development and enables non-programming experts to become effective programmers within a domain-specific environment.

Programming support tools have been developed in the LOLITA development environment. These tools offer information to a variety of multi-skilled users. They

[9]The Chalmers Haskell Compiler (HBC) and the Glasgow Haskell Compiler (GHC) interface files in particular.

also offer information about the practical complexity of the natural language system, an issue which is paramount in the development of a successful real-world system within the domain of Natural Language Engineering.

3.7 Chapter Summary

This chapter deals with functional programming on a large-scale. A number of real-world functional programmed systems were introduced, supported by the results of the Dagstuhl workshop on Functional Programming in the Real World. The distinction between a Related Real-World System and an Unrelated Real-World System was made.

The majority of the Unrelated Real-World Systems cited were written in a version of the ML language; of those which were written in Haskell the LOLITA system was the largest.

The LOLITA Natural Language processing system is currently being developed at Durham University. It consists of 50,000 lines of Haskell code. It has 20 developers who vary in their functional programming ability. A large percentage of these developers are not Computer Scientists, some have had no programming experience prior to joining the development team.

There are a number of features of functional languages which have been adopted to make programming the system an easier task. The creation of domain-specific sub-languages, together with higher-order functions and currying, has enabled a number of levels of programming to be created in LOLITA. Languages have been built which support the problem domain, examples were demonstrated with the LOLITA post-parse stage of analysis and the LOLITA generator.

Our experience of developing a large-scale application in Haskell has allowed a number of improvements to be made to the Haskell language and specifically to the Glasgow Haskell compiler. The work has also determined the introduction of methods enabling the successful development of an unrelated real-world system.

Chapter 4

Profiling LOLITA: Case Studies

4.1 Introduction

This chapter is based on a record of profiling experiments carried out on the LOLITA system over a two-year period. The Haskell profiling tools used in these experiments were the Runciman and Wakeling heap profiler [Runciman and Wakeling, 1993] for the HBC compiler and the Sansom and Peyton Jones cost-centre profiler [Sansom and Peyton Jones, 1995] for the GHC compiler; section 2.6.2.

The cost-centre profiler was used to measure time and memory usage. The heap profiler was used to record memory usage.

Information about memory usage is of particular interest to members of the LOLITA development team; storage management is not controlled directly by the programmer and a seemingly innocuous change to a function during development can make a big difference to the amount of memory used. The heap profiler displays the amount of heap space used by each function, constructor or module, as a serial profile graph; in this way a developer can see how much heap is consumed as the execution of LOLITA progresses.

Both time and serial time profiles are useful in the development of an efficient system. The constraint on execution time is not as rigid as that on heap usage. A program will fail if it runs out of heap space, but it is possible to wait longer for

an execution to complete. This said, it is essential that the system remain time efficient if it is to meet its NLE specification, see section 3.3.1.

The profiling experiments had three main objectives. The first was to improve the efficiency of the LOLITA system, creating a system which would require less space and time to execute, and which would nevertheless retain its modularity, readability and maintainability.

The second objective was to identify and experiment with a number of methods used in the profiling of a large system. Attempts have been made to identify different profiling methods and requirements. Clayman, Clack and Parrott make the distinction between application programmers and system implementors, section 2.3.3 [Clayman, Clack and Parrott, 1995]. This distinction is coarse and seemingly based on conjecture. Sansom and Peyton Jones by contrast experiment with a top-down approach in profiling the Glasgow Haskell Compiler, section 2.6.2 [Sansom and Peyton Jones, 1994]. The case studies carried out on the LOLITA system allow us to study what developers do when profiling an unrelated real-world Haskell system.

Finally the experiments aimed to identify any weaknesses which might exist in the current profiling tools, and any enhancements or new tools which might improve the efficiency analysis of the system. By studying maintainers and developers in action, it is possible to study the specification of both existing and new tools.

4.1.1 Managing the case study information

Such a study was clearly going to produce an enormous amount of information; the LOLITA system contains a large number of operations and once the performance of these operations had been recorded, changes would invariably be made to the code. The performance of the system would then be re-analysed and the resulting improvements collected. The changes made to the system during this profiling cycle would provide valuable results for the study; however, recording information on such a large scale was impractical.

It was clearly important to develop criteria which would represent the changes and decisions made during the profiling task. This would avoid having to store a large number of data files and different versions of the system, and would also structure any desirable reconstruction.

The following machine-based parameters were recorded:

- Haskell compiler options used — This includes compile time flags; the optimisations used `-O*`; heap and stack sizes `-H` and `-K`; the garbage collector used `-gc-*` and other options including compiler extensions, C pre-processors and interfaces.

- Haskell profiler options used — This includes profiler flags; creating cost centres automatically `-prof -auto-all`, or by annotating the code `-prof`; and the method of profiling, `-pT/A/C` and `-P` for time profiling, `-hC/M/G/D/Y/T`[1] for heap profiling.

- Machine used to perform experiments — Although this should not affect the profiling results to any great extent, subtle differences have been experienced between different machines[2].

- Output from the the profiler — The raw results including tables and graphs, and the directories and files in which these results are stored.

- Run-time options — Exactly what is being profiled. This includes the LOLITA command which is executed, for example 'parse', and the data given.

- Version of source code — Each file in the LOLITA system has a version number; these were recorded. The system data used was also recorded, as different data sets would clearly affect the results. Modifications to the source code and data were also recorded, so that progressive changes could be monitored.

[1] These flags are based on GHC. Similar HBC flags are also available.

[2] For example when switching between a SPARC Centre 2000 with 6 cpus, 256Mb of memory running Sun OS 5.3 with a swap space of 1 Gb, and a SPARC ELC with 64 Mb of memory and 200 Mb of swap space running Sun OS 4.1.3.

Parameters were also recorded which showed the reasoning behind various decisions made during profiling. For example, why a particular change was made to a section of code, or why the programmer's attention focussed on a module, function or expression. This allowed the behaviour of the programmer and the decisions taken to be understood in response to a set of profiling results.

4.2 Case study I: Low-level functions

4.2.1 Aims

Some profiling tools decide independently which parts of the program are profiled, so the starting point of profiling a program is predefined.

Those profiling tools, such as the cost-centre profiler, which allow the programmer to decide which parts of the program they should investigate during profiling, make the profiling starting point more uncertain. The programmer can either annotate functions which he is interested in profiling, or alternatively the programmer can profile the whole program and begin the investigation based on these results.

The first case study investigates the time profiling of LOLITA using the cost-centre profiler. It begins by profiling the complete system, utilising a facility in the cost-centre profiling tool which allows all top-level (globally-defined) functions to be profiled. This facility is specified at compile time using the `-auto-all` compilation flag. A flat profile is produced.

Profiling all top-level functions in the system allows the profiling to begin with the assumption that the user is not an expert in the LOLITA code or with the system functionality. An alternative and equally valid starting-point would have been simply to annotate the top-level system functions, such as the parsing, semantic analysis and generation functions. The latter approach would require the programmer to identify those important components of the system, a process which would be prone to errors without a detailed knowledge of the system.

An overall profile of the system also demonstrates what sort of results the automatic profiling produces.

4.2.2 Analysis

Run-time data and operations were carefully chosen to utilise a large number of system operations during the execution of the LOLITA system. The LOLITA template analysis task is well-suited to profiling, as it utilises most of the core operations of the system.

When the LOLITA system is executed, the internal semantic-net structure is loaded. This operation makes heavy use of the `readvals` function. Developers of the system are aware that this operation is expensive, yet it is a one-off cost. It is the execution costs of the system after this set-up which are more interesting to the programmers.

The profiling results are displayed as a percentage of the total run-time costs. Therefore, to avoid the system set-up skewing the results of the template analysis, the execution of the program during template analysis must be long enough to stabilise the results. This is done by balancing the number of executions of template analysis against the time required to perform the system set-up.

Profiling the system in terms of all the top-level functions is likely to show the basic core functions which are called thousands of times. Each execution of these functions is unlikely to use much system time, but the reliance of many of the LOLITA operations on these core functions will mean that they are utilised widely throughout the system, and hence account for a high percentage of the total execution time. The early results, some of which are shown in Figure 4.1, show that this is indeed the case.

By highlighting these functions it was possible to optimise these small portions of the code, creating initial improvements of 7.8%. This was surprising, as no detailed algorithmic analysis was needed—simply recording the number of times each function was called and the total execution time of each was sufficient.

```
Tue Feb 22 11:12 1994 Time and Allocation Profiling Report (Final)
           (Lexical Scoping)

   lolita.exec +RTS -pT -H230000000 -RTS

total time  =      381.44 secs   (19072 ticks @@ 20 ms)
total alloc = 464,314,036 bytes  (34922110 closures)
```

COST CENTRE	MODULE	GROUP	scc	subcc	%time	%alloc
updateElem	Stdenv2	Stdenv2	7225	0	9.6	6.5
member	Stdenv	Stdenv	83	39780	4.7	7.9
readvals	Stdenv2	Stdenv2	4	516	3.8	14.0
hash	Hashdict	Hashdict	22154	0	2.4	3.5
hashin	Hashdict	Hashdict	1	0	2.1	4.4
linkob2	Total	Total	1	0	2.1	1.6
dot2	Stdenv2	Stdenv2	6	9988	1.5	3.4
indexLA	LArray	LArray	195230	390460	1.5	1.5
flat_order	Stdenv2	Stdenv2	232335	151000	1.4	4.5
sel_meanings	A_meaning	A_meaning	49880	106390	1.4	1.6
.						
.						
CAF:ds	LArray	LArray	0	0	23.2	0.0
CAF	Prelude	Prelude	0	0	5.5	10.7
CAF:ds	Stdenv2	Stdenv2	0	0	4.3	0.0
DATA	Prelude	Prelude	0	0	1.3	0.0

Figure 4.1: Example of the initial profiling results.

Improvements to the system at a low level manifest themselves as overall improvements simply because they are called a large number of times.

Despite initial improvements to the LOLITA system, this method of profiling is limited. Consider the following example.

The profiler displayed the `member`[3] function as having a high proportion of system costs. Despite the fact that the function was only five lines long, it recorded nearly five percent of the system costs. An improvement to the definition of this function had potential benefits.

The code for the `member` function, however, appeared efficient. In this case,

[3]This is a LOLITA membership function and not the Haskell prelude membership function.

therefore, the low-level approach was simply not enough to enable any improvements to this part of the system; further information was needed to interpret the results and develop an understanding of the problem. The `member` function is a generic piece of LOLITA code used throughout the system and analysing the efficiency depends on the way in which it is used, rather than simply on the quality of its algorithm.

From experience, it was known that one of the places in the system where the LOLITA generic membership function is used is in the classification of proper nouns. With over two hundred different word endings the task of checking the ending of each word parsed with this list is a lengthy task. The use of the LOLITA generic membership function in this example means that the search is *O(n log n)*.

A significant improvement to the time spent in the membership function was produced by re-implementing the list of word endings as a tree, thus reducing the search time to *O(log n)*.

4.2.3 Concluding remarks

In this example we were able to apply our knowledge of the system to the problem. The profiling tool was clearly useful in identifying the general problem, but without any in-depth knowledge of the system a solution was not obvious. Essentially the list of word endings needed re-implementing and separating from the LOLITA generic membership function. A simple re-programming of the membership function would have had no effect.

The membership function is shared between many higher-level functions. To identify which of these higher-level functions was responsible for a large proportion of the costs, it would have been necessary to add cost centres to all the places where the membership function was called. This process would then have had to be repeated on the higher-level functions until the path of expensive calls to the member function was identified.

It is noted that changing the cost centres in the code requires re-compilation

of those modules in which the changes are made. This method of moving the profiling costs to higher-level functions in the program could prove a lengthy task. In this case an accurate method of cost inheritance, (alongside the flat profiles produced by the cost-centre profiler), would have been useful. It would then have become obvious to the programmer which calls to the membership function were particularly expensive.

Post-processing a single set of results rather than re-compiling and re-running the program a number of times would have also allowed these results to be collected in less time.

4.3 Case study II: Grammar transformations

4.3.1 Aims

The second case study investigates a memory problem experienced in the development of the grammar transformations in the LOLITA system. The grammar transformation program takes the grammar of the LOLITA system and applies a series of transformations to it, producing a new grammar as the result.

It was clear that such a process was going to take a reasonable length of time, though once fully operational it could be installed on-line as part of the compilation process and need only be run once. If any of the grammar rules were changed then a re-compilation would be necessary in any case.

During development the issue of how long the transformation process took to complete was not of primary concern. However, these transformations seemed to be consuming large amounts of memory to the extent that there was a shortage of memory while the transformations were taking place. An investigation ensued using the GHC heap profiler to discover why so much heap space was being used.

4.3.2 Analysis

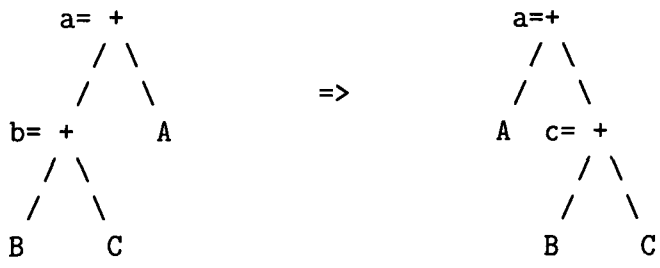
The transformation grammar is represented as a graph and the transformations performed on it are essentially operations on segments of the graph. It was clear that one of the major costs involved in the process was the storing of the graph itself, which was implemented as an array. It was possible to determine how much memory space the graph required by performing calculations on the number of nodes stored in the graph. The results of the transformation process showed that the memory consumption far exceeded our expectations.

The investigation began by producing a heap profile of the operation. The transformation process had initially run out of heap space before reaching the end of the grammar files, therefore the data set was reduced for the first set of profiling experiments. By profiling the grammar transformations with a single grammar file, rather than a collection of files, the results could be examined on a smaller scale and then scaled up.

The initial profiling results highlighted the function `applyToAnyGood` in the `TranEngine` module. This function simply applies the given transformation to any node in a list of so called 'good nodes', returning the first successful transformation result (or alternatively failing).

```
> applyToAnyGood:: ShowTransF -> [([Char],Transform)] -> Graph ->
>                  TraceM (Bool,Graph)
```

As well as returning the new graph, a pool of new 'good nodes' is calculated by subtracting from the old 'good nodes' any of the non-terminals matched against constructs on the left hand side of the transformation, and then adding all new or updated non-terminals from the right hand side of the transformation. For example, in a simple *'normalise or'* transformation,



$[a, b]$ are removed from the pool of good nodes and $[a, c]$ must be added, since they are referenced in the new graph. The net effect is that b is deleted from the pool and c is added, whilst a remains in it.

The function `applyToAnyGood`, in Figure 4.2, is large, and although the profiler had identified this function as being the main consumer of large amounts of space, it was still necessary to pinpoint exactly where in the function the problem occurred. Cost centres were added to the function by hand to reduce the function into a number of expressions, with the aim of finding the exact location of the problem in the function. The definition in Figure 4.2 includes the cost centres added by hand. The heap profiling graph in Figure 4.3 shows that the added cost centres reduce the heap problem to the code identified with the `stats` cost centre.

The second heap profile graph in Figure 4.4 shows the constructor profile for the same execution; the list constructor : is clearly the main user of the heap[4].

The section of the function identified with the `stats` cost centre builds a string of statistics about the particular transformations, indicating how much of the transformation process has taken place. Its result is simply a list of characters. So the part of the program where the efficiency problem manifests itself has been identified, but this is only half the battle. Once the problem code has been identified it is then necessary to calculate exactly what is wrong with the code and how this efficiency problem can be solved.

The immediate reaction to these results was that the problem might be in the append file operation. The statistics produced by the transformation process were

[4]It is noted that the investigation uses full-heap producer and constructor/closure profiles and not profiles of selected slices of the heap. The latter method is an alternative and equally effective method of profiling.

```
> applyToAnyGood st ntrs g
>  = apTAG (ls (nonterms g)) g
>    where
>     apTAG good g
>       | (good == emptys) = scc "emptyTrace" (emptyTrace (False,g))
>       {- | is_And nt_node && member nt_left_children nt
>          = error
>              ("cycle found\n"++showgraphs g nt_left_children) -}
>       | trs/=[]
>         = (if nt==9 && fst (head ntrs) == "Complex Unify [2]"
>           then trace ("UNIFY 9, ORS: "++show or_children++"\n\n"++
>                       showgraph g)
>           else id) $
>           mapTR (apply_fst (const True)) $
>           addTrace (st (fst (head trs)) mg mnts g' rnts ++stats) $
>           apTAG new_good (clear_pool g')
>       | otherwise = {- addTrace ("FAILED nt= "++shownonterm nt ++
>                                  "new_good= "++shownum
>                                               (card new_good')) $ -}
>                     apTAG ({- traceFun (\s -> "apTAG failed: "++
>                                               show good++
>                                         "\n new = "++show s) $ -}
>                     new_good') g

>         where
>           nt = hd_set good
>           trs = [(tn0,tr) | (tn0,tr0)<-ntrs, tr<-applyTopTrans
>                             tn0 tr0 g nt]
>           ((mg,mnts),(g',rnts)) = scc "mgmntsgrnts"
>                                   (snd (head trs))
>           new_good = ls (get_pool g') `sunion` good `sdiff`
>                      (ls mnts `sdiff` units nt)
>           new_good' = tl_set good `sdiff` unmatchableNodes
>                                   g nt ntrs
>           or_children
>              = sort (orChildren g nt [])
>           stats = scc "stats" (" nt="++shownonterm nt++
>                   " new_good="++
>                   {- show (take 10 (sl new_good))++ -}
>                                 shownum (card new_good)++
>                   " size="++shownum (get_size g')++
>                   showTrans ntrs (mg,mnts) (g',rnts)
>                   )
The following code checks for or level cycles of a special type:
>           nt_left_children = leftChildren g
>                                 [head (subts nt_node)] []
>           nt_node = snd (get_node g nt)
```

Figure 4.2: The function applyToAnyGood.

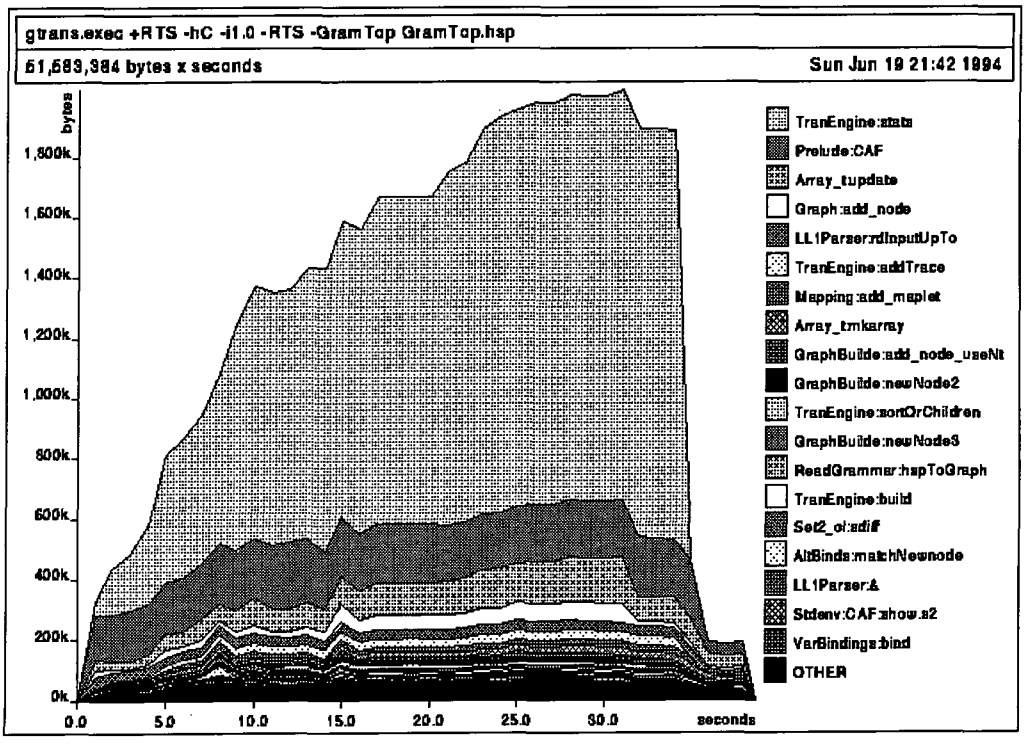


Figure 4.3: First grammar transformation profiling result.

stored in a file (in fact in this instance the `appendChan` was used to write the results to the standard output). It was possible that appending these results to the file was causing some sort of space leak; that is, when the program was appending the results to a file it was not discarding part of the graph.

The `appendChan` code was analysed and modified so that the suspected bug was removed. The system was recompiled and the profiler executed once more on the transformations. The results of this execution are displayed in Figure 4.5 and shows that this first attempt at a fix was unsuccessful.

The code needed further analysis and a re-investigation was required. The `stats` identifier was the source of a second lead, as it was also used as part of a trace.

The *LOLITA* system has a separate module for a trace which implements the programmer's own version of the tracing function. This tracing function is implemented using a monad:

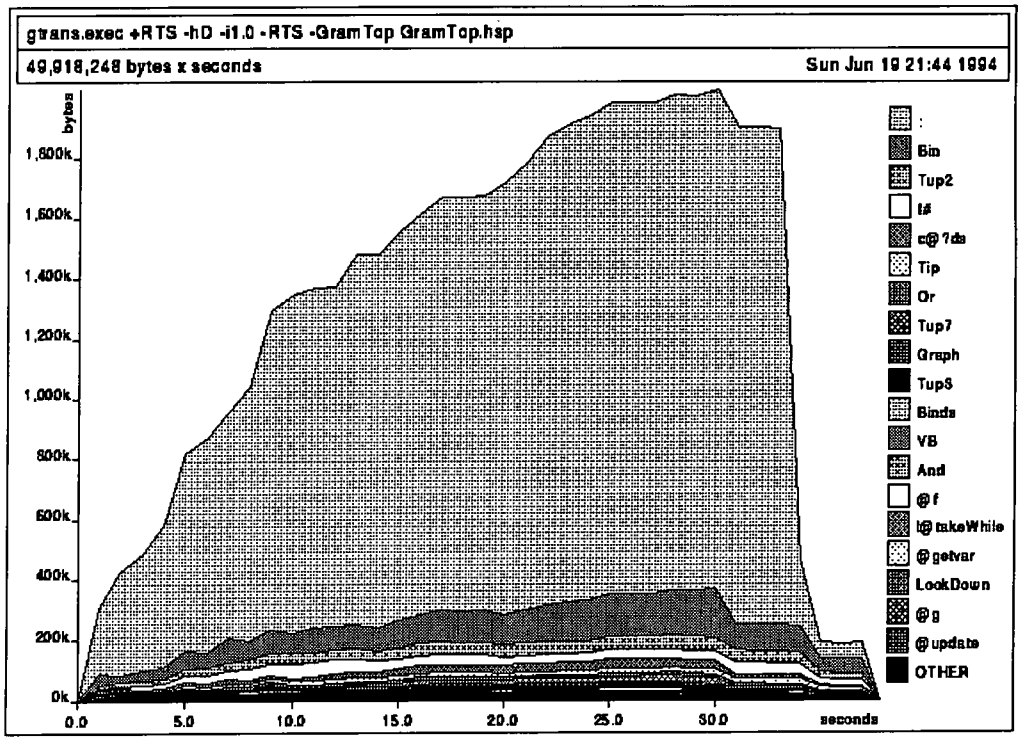


Figure 4.4: Second grammar transformation profiling result.

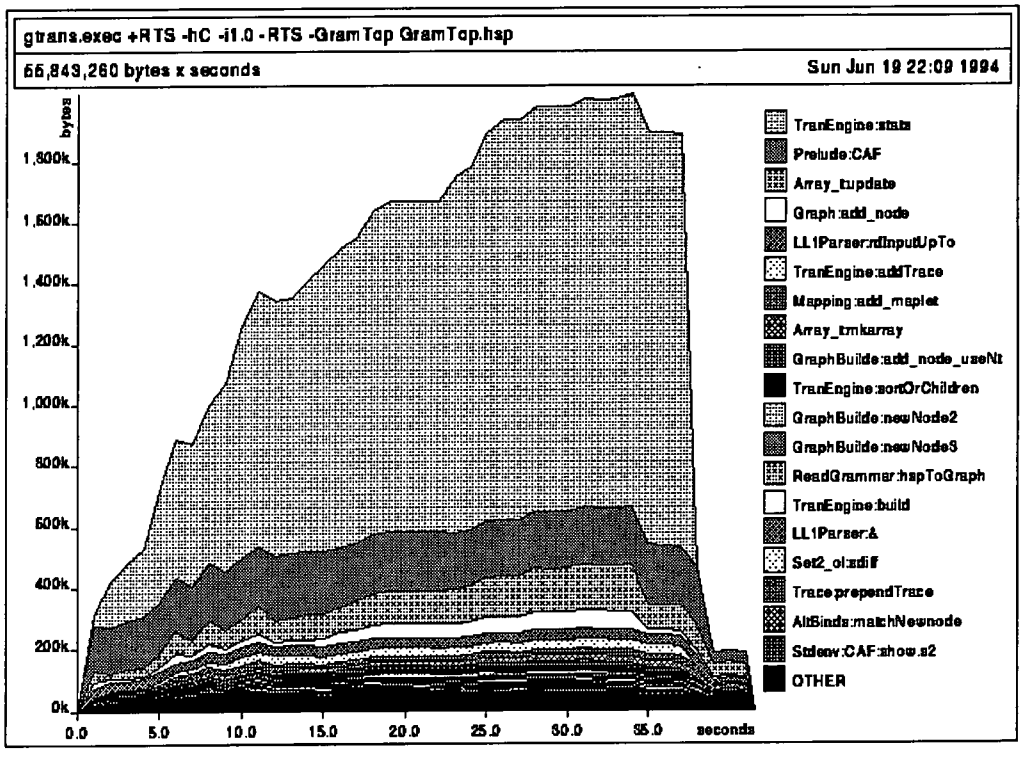


Figure 4.5: Third grammar transformation profiling result.

```
> emptyTrace:: a -> TraceM a
> thenTrace:: TraceM a -> (a -> TraceM b) -> TraceM b
> thenTrace_:: TraceM a -> TraceM b -> TraceM b

> traceMsg:: [Char] -> a -> TraceM a
> addTrace:: [Char] -> TraceM a -> TraceM a
> getTraceRes:: TraceM a -> (a,[[Char]])

> data TraceM a = TM a [[Char]]
> selTrVal (TM a tr) = a
> selTrMsg (TM a tr) = tr

> emptyTrace x = TM x []
> thenTrace  (TM a tr) tf = prependTrace tr (tf a)
> thenTrace_ (TM a tr) tf = prependTrace tr tf
> traceMsg m v = TM v [m]
> addTrace m tr = TM (selTrVal tr) (m:selTrMsg tr)
> getTraceRes (TM v ms) = (v,ms)
> prependTrace xs tr
>  = TM (selTrVal tr) (xs ++ selTrMsg tr)
```

Further analysis ensued and a problem with this version of the trace monad was identified. The program outputs a trace as it runs but the trace contains not only the trace but also the associated piece of graph that is being transformed. Ideally when the trace is output the piece of graph should be discarded; however the implementation of the program held onto this trace, and the associated section of graph, each time it was executed. Only when the whole expression had been evaluated could all these trace/graph pairs be discarded. The profiling results seemed to support this hypothesis.

Interestingly the implementation was not as lazy as we had hoped; the trace monad needed redefining:

```
> data TraceM a = TR String (TraceM a) | TV a

> emptyTrace x = TV x
> thenTrace (TR m tr) tf = TR m (thenTrace tr tf)
> thenTrace (TV a) tf = tf a
> thenTrace_ (TR m tr) tf = TR m (thenTrace_ tr tf)
> thenTrace_ (TV a) tf = tf
> traceMsg m v = TR m (TV v)
```

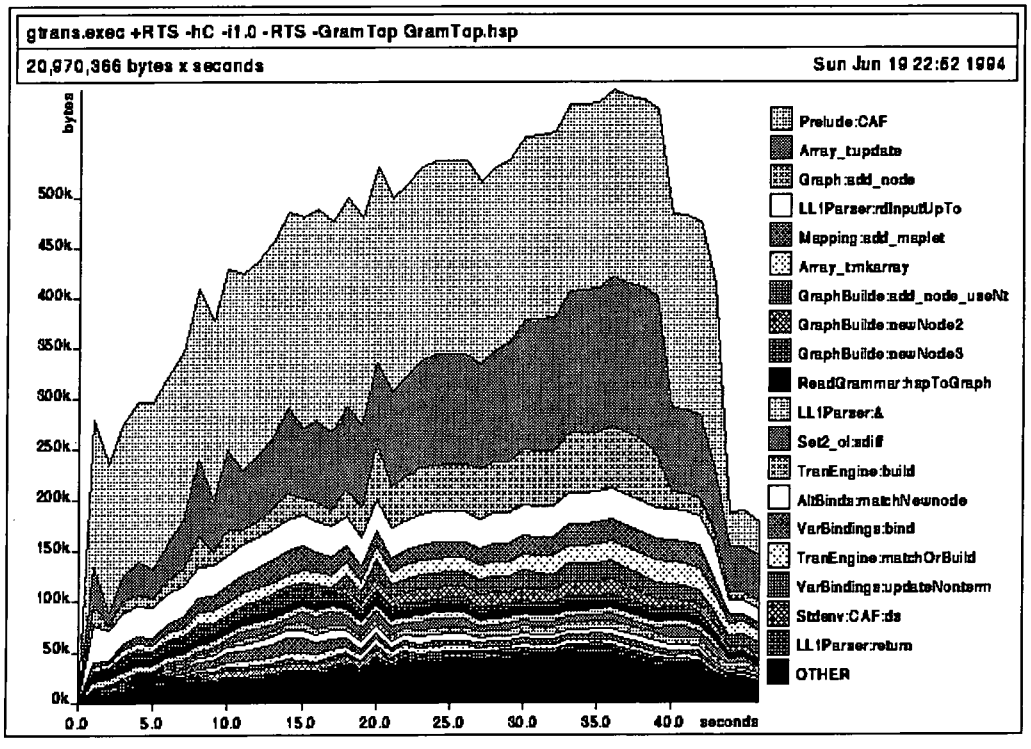


Figure 4.6: Fourth grammar transformation profiling result.

```
> addTrace m tr = TR m tr {- Note addTrace has been modified -}
> getTraceRes (TR m tr) = (v,m:ms)   where (v,ms) = getTraceRes tr
> getTraceRes (TV v) = (v,[])
```

In Figure 4.6 the results show that this new trace monad was far more successful than the first; it more than halved the heap usage.

The results in Figure 4.7 show the execution of the grammar transformations on the full data set of all the grammar files. The results show the heap space levelling out to become constant over the last 30 seconds of the execution. Figure 4.8 shows this execution extended to 180 seconds of execution time. As expected, the increase in the space used by the cons constructor simply reflects the increase in the size of the transformation graph. The heap problem was solved by removing the space leak in the trace monad, allowing more of the grammar transformations to be executed and reducing the heap space required.

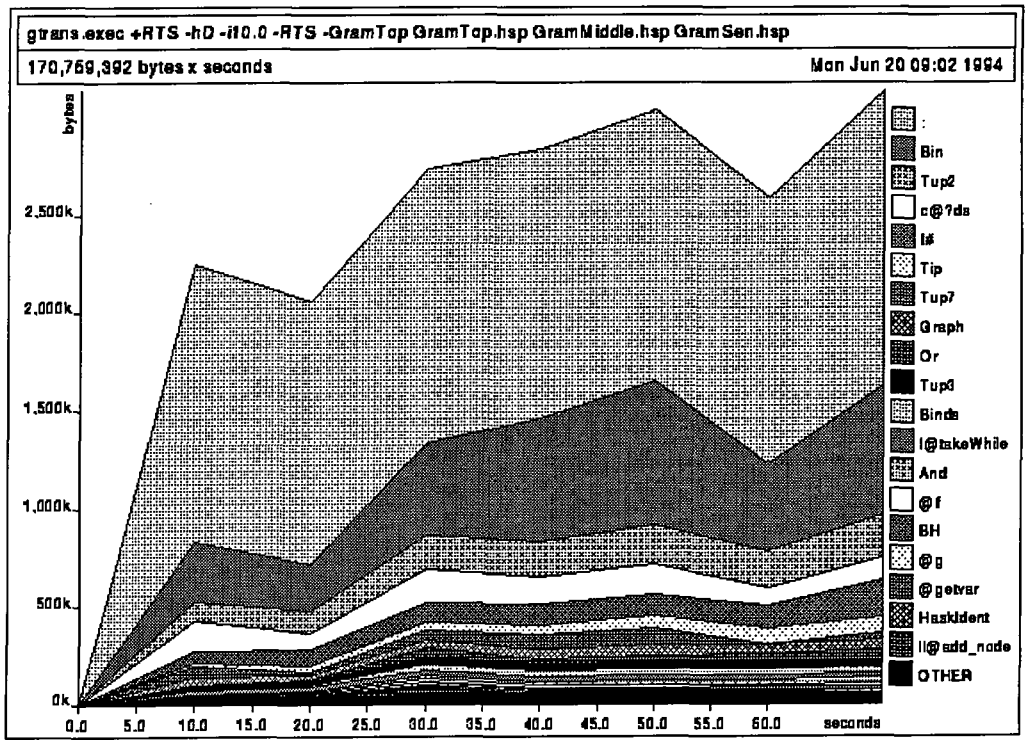


Figure 4.7: Fifth grammar transformation profiling result.

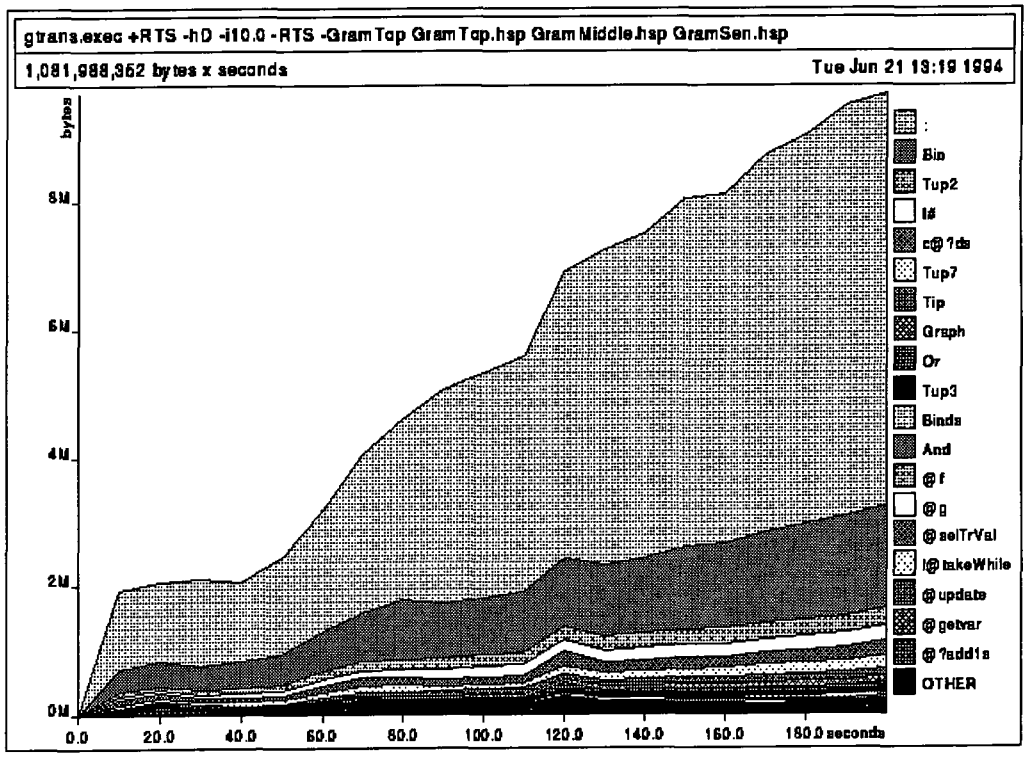


Figure 4.8: Sixth grammar transformation profiling result.

4.3.3 Concluding remarks

Using the profiler in this example does eventually allow the problem to be identified. The profiling results do show that storing the statistics generated by the grammar transformations is causing the problem in the code, but they do not indicate, which part of the code is responsible for retaining the space which is a further problem to be solved.

Such results point to the usefulness of the work being carried out by Runciman and Röjemo on the retainer profiler [Runciman and Röjemo, 1996], section 2.6.2.

The cost-centre profiler is limited in this respect. Annotating every function in the trace module would have narrowed the problem down to an individual function. However, it would not have indicated that the space was caused by holding onto parts of the graph and trace string specifically, or what piece of code was responsible.

4.4 Case study III: The transformation engine

4.4.1 Aims

The third case study follows the continuing development of the grammar transformations. The grammar transformations are divided into a number of sections and the initial problems with the trace monad were experienced in the first stage of these transformations. After fixing the space leak in the trace monad the grammar transformations were proceeding much further than they had before, but at the second stage of the transformations further problems were identified.

Figure 4.9 demonstrates a massive improvement on the first set of grammar transformation profiling results; the first stage of the transformations with the complete set of grammar transformations finishes after 250 seconds. It is at this point in the profile graph, as the second stage of the transformation begins, that further problems become apparent. Figure 4.10 shows this problem broken down according to the program constructors.

4.4.2 Analysis

Attention was drawn to the function `matchOrBuild`, the fifth function down in the profile graph of Figure 4.9 and defined in the module `TranEngine`. The results in Figure 4.10 show the constructors involved in building this part of the heap; the `add-node` function closure and the `Or` constructor are particularly worrying.

The heap size increased to give the heap profile shown in Figure 4.11. The larger heap size enables profile graphs to be produced which imply that the problem must be caused by the `matchOrBuild` function. Figure 4.12 shows the constructor/closure profile of the same problem, confirming initial worries about the `add-node` function closure and the `Or` constructor.

The `add-node` construct adds new nodes to the transformation graphs. Initial conclusions were that it was this construct and a desugaring which were causing

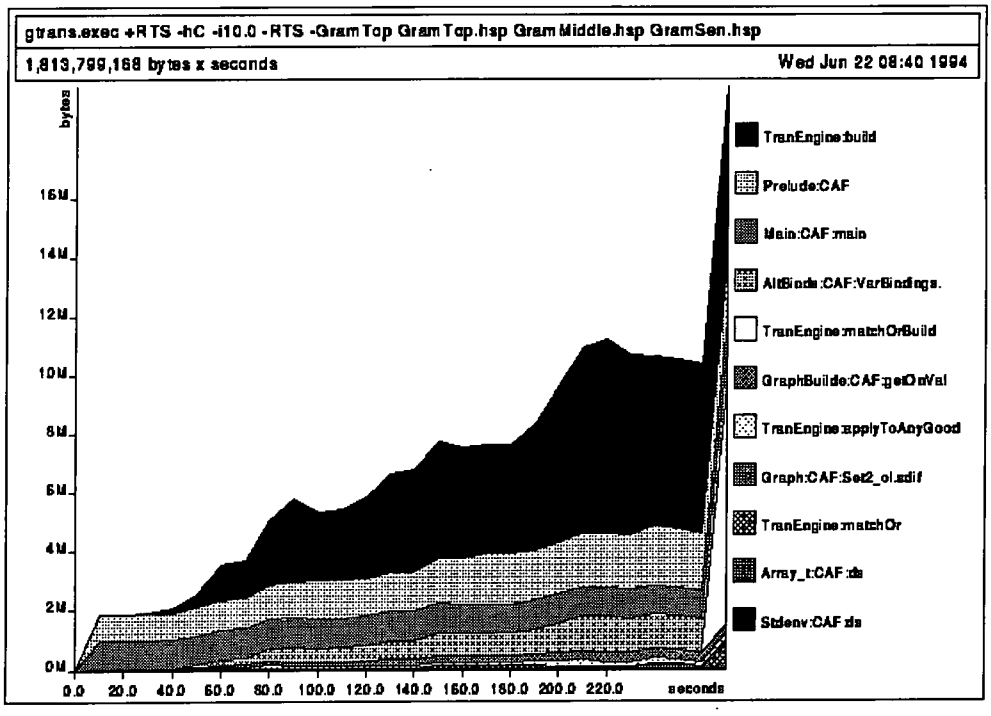


Figure 4.9: The first stage of the grammar transformations are complete by 250 seconds of execution. At this point in the profile graph further problems become apparent.

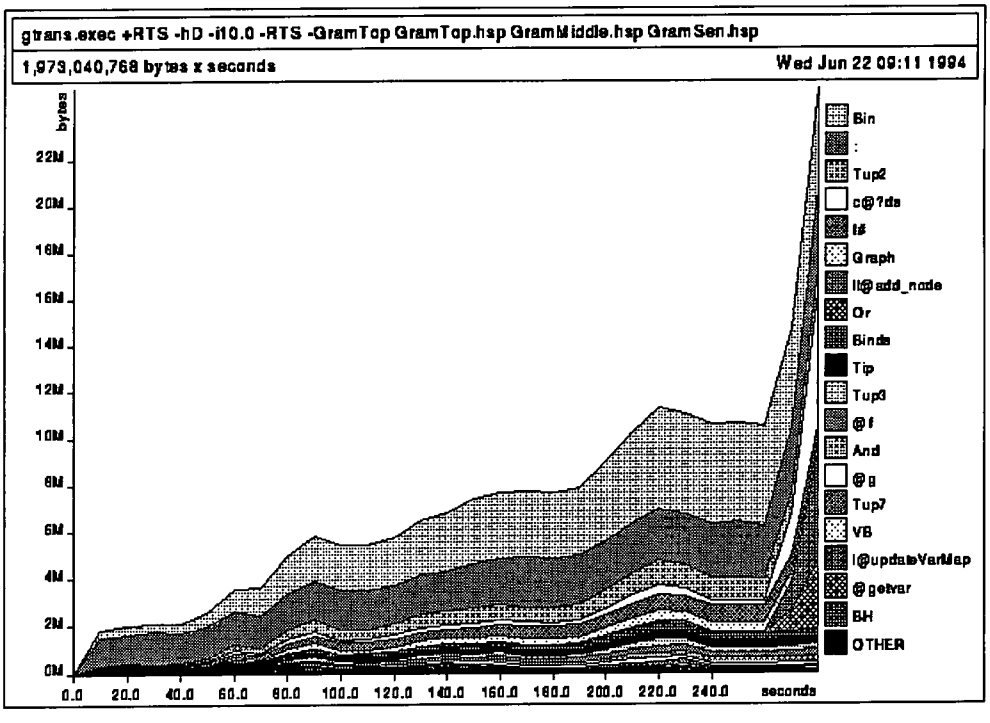


Figure 4.10: The heap problem after 250 seconds of execution time, broken down according to its constructors.

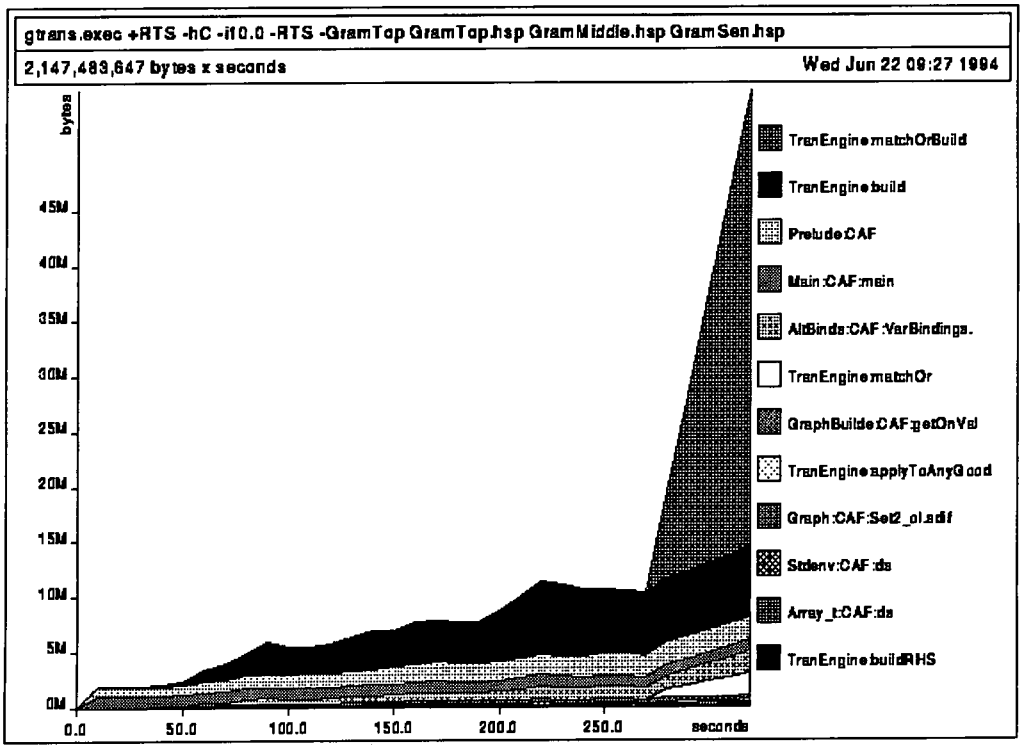


Figure 4.11: Increasing the heap size to extend the length of the serial heap profile to $\approx$ 300 seconds.

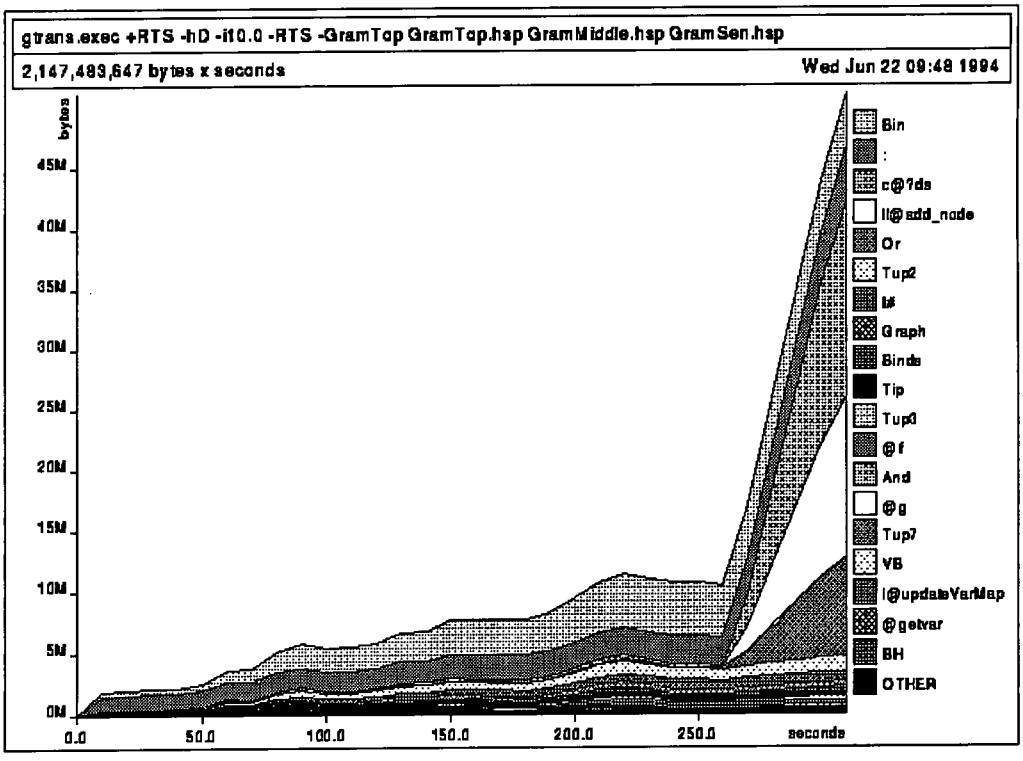


Figure 4.12: The results of the extended serial heap profile shown in terms of the constructors.

the increase in heap usage. The `add-node` function was strictified to ensure that it completely updated the graph. Some other small program improvements were made and the code was profiled once again.

The next profiling graph showed no improvements in the heap usage, indicating that this was a false alarm once again. (Since this did not solve the problem with the code the profiling output has not been included.) The vast amount of heap space (around 50Mb) was still being used by something and there was little idea at this stage what it was. Continuing the analysis the `matchOrBuild` function was also strictified.

```
> matchOrBuild m [] nt     =
>                    matchNt m nt
> matchOrBuild m (o:os) nt =
>                    matchOrBuild m os `matchNewnode` (Or o nt)
```

This did not make any difference to the profiling results either, so an `scc` expression was added to the function `matchNewnode` in the alternate variable bindings module `AltBinds` which uses the grammar graph. This narrowed the problem down further, see Figures 4.13 and 4.14, but did not go any further towards identifying what the problem actually was.

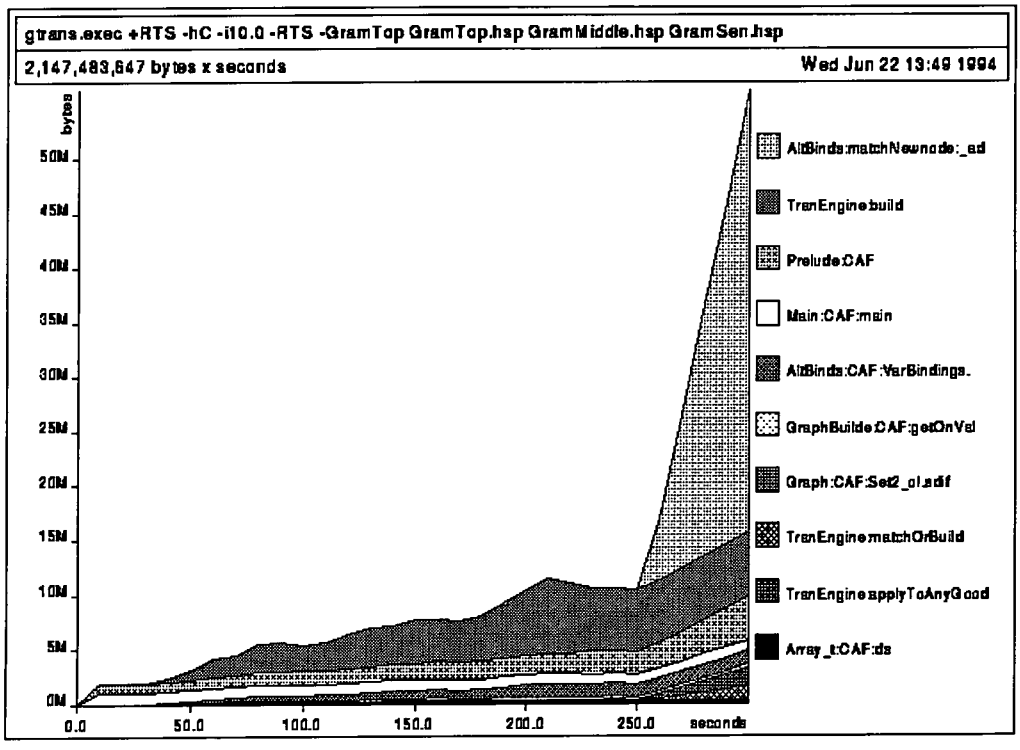


Figure 4.13: The results of an `scc` expression to the function `matchNewnode`.

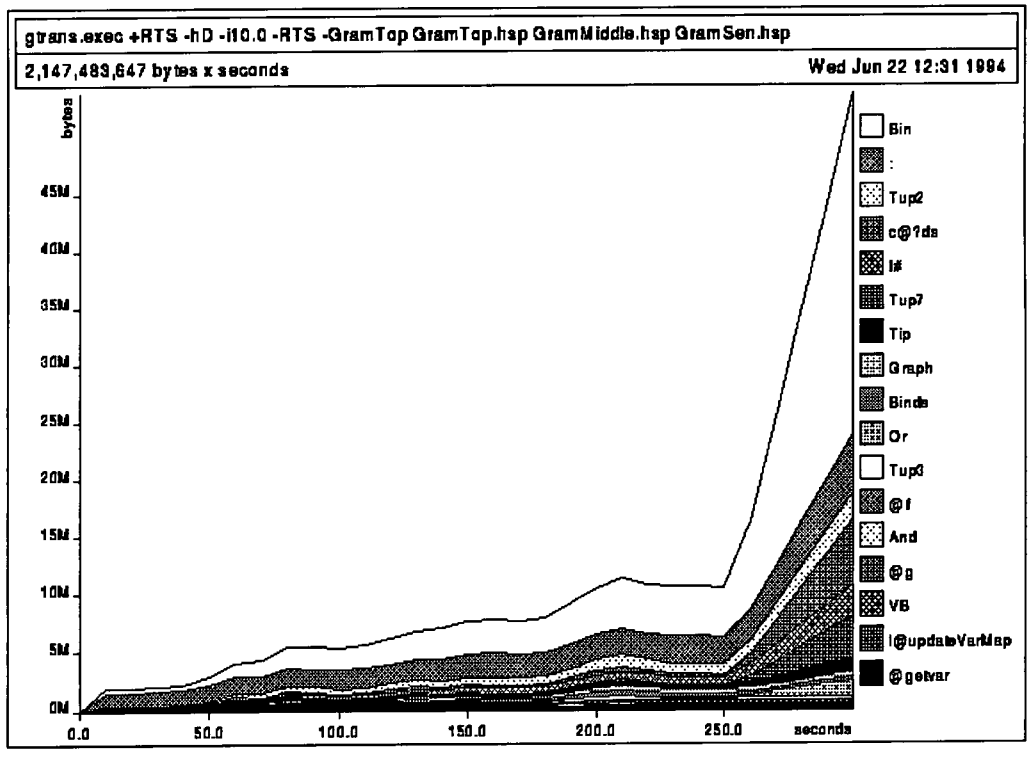


Figure 4.14: A constructor profile of the same program.

```
> matchNewnode f n
>  = AB (concmap f_b)
>    where
>      f_b ((g,ms),b)
>        = selAB (f nt) [((g',ms),b)]
>          where
>            (nt,g') =
>              add_node_lazy AllowGC (getAllNts b++ms) ("",n) g
```

The fact that strictifying these functions was making no difference to the overall heap usage suggested that these functions were having their evaluation forced by a parent function. The process of analysis was therefore altered to consider the behaviour of the parent functions of `matchNewnode` and `matchOrBuild`. Experience of this part of the system helped the analysis and soon it was possible to trace a sequence of function calls back to the function `applyTopTrans` in the `TranEngine` module. This function seemed to be the cause of the problem.

```
>  applyTopTrans g s t nt
>   = trace ("applyTopTrans (attempt): "++show nt) $
>     let res = applyTrans g t [GSform nt]
>     in if null res
>         then {- trace "applyTopTrans failed" -} res
>         else {- trace ( "applyTopTrans count: "++
>                 show (get_size $ fst $ fst $ head $ res)) -} res
```

The complete set of transformations was being evaluated because the length of the list was calculated before the first element of the list was chosen. The erroneous definition of the function, shown above, shows that the last line of the function calculates the size of the remainder of the list, with, as it turned out, disastrous results.

The final profiling graphs show an impressive improvement to the grammar transformation program and the massive saving in space, around 50Mb in the first 250 seconds of execution(!), which these changes made.

This is an interesting example; the profiler again shows that a problem exists but does not identify its cause. The process of profiling and improving the grammar transformations program took a total of four days to complete.

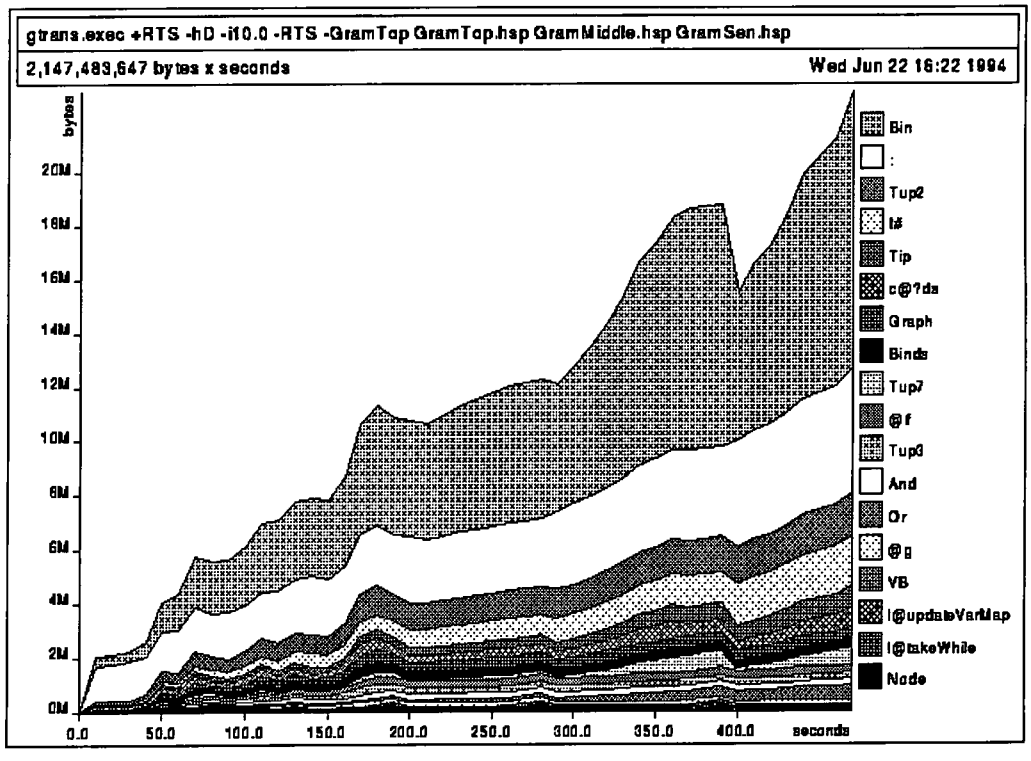


Figure 4.15: A constructor profile showing the huge improvement to the grammar transformation program with a corrected version of the `applyTopTrans` function.

4.4.3 Concluding remarks

As has been seen, the profiling results in this case study, though not entirely misleading, made the task of identifying the problem in the code very difficult. This highlights a fundamental problem which is illustrated in the functional dependency graph for this portion of the LOLITA program (over page). This graph shows the relationship between the profiling results, bringing to the attention of the programmer the `matchOrBuild` function, and the function where the problem in the code actually was, in the `applyTopTrans` function.

It could be argued that placing cost centres higher and higher in the call-graph would have eventually found the problem but this is not the case. Propagating costs up the call-graph would not in fact make any difference and the large costs displayed at the `matchNewNode` level would also be displayed at the higher levels, culminating eventually at the `main` function.

```
applyTopTrans (TranEngine module) Where the problem in
    |                             the code actually is
    |
applyTrans    (TranEngine module)
    |
matchPars     (TranEngine module)
    |
matchPar      (TranEngine module)
    |
matchNt       (TranEngine module)
    |
match         (TranEngine module)
    |
matchOrNC     (TranEngine module)
    |
matchOr       (TranEngine module)
    |
matchOrBuild  (TranEngine module) Where the profiling
    |                             results are displayed
    |
matchNewNode  (AltBinds module)   Adding cost centres
                                  presents the costs at
                                  a lower-level
```

When using cost centres it is usual to work down in the call graph to lower-level functions in a top-down process.

```
function a
    |
function b
    |
function c
```

If a large cost were displayed at level a, then these costs could be traced by further annotating functions at the next level down in the function dependency graph, at level b. This would continue to level c, until the problem in the code was narrowed down. However, if the problem was displayed at a low level then with a shared function it would not be easy to move the costs up the function dependency graph without knowing in which direction to head.

Improving the situation would require a method of profiling which could inherit costs to parent functions. The cost-centre profiler provides flat profiles and therefore makes no attempt to inherit costs to higher-level functions in the call graph. Implementing a statistical approach to inheriting costs may provide an adequate solution to this problem in a number of simple cases, though it also has the potential of making the results even more misleading if functions are heavily shared.

Accurate inheritance would allow the programmer to follow the costs back through the program to the point in the program where the offending code appeared. In this case study a method of accurate inheritance profiling may well have helped identify the problem quicker. It would certainly have helped to narrow down the search.

The length of time that this analysis needed is also open to improvement. Changing code and re-writing cost centres in the code requires re-compilation. This time can quickly add up and make profiling frustrating. A post-processing scheme of manipulating profiling results after program execution would reduce the time needed for profiling.

It seems feasible that the efficiency of the program could be monitored in a single execution and the programmer could then decide, at a later date, in terms of which functions or expressions he wanted the results displayed. This would allow the programmer to speed up the profiling process and perhaps be more adventurous with the analysis of the program. Such schemes are developed in chapter 5.

4.5 Other Case Studies

The case studies presented have been chosen from a set of data collected over a period of two years. They illustrate some of the recurring problems experienced during the profiling of the LOLITA system.

There are a number of general points which can be made regarding other case studies which have been carried out and whose results are not included in this thesis. They highlight potential hazards with the cost-centre profiler and the HBC heap profiler and offer ways in which these tools may be improved.

Functional programs have a tendency to heavily reuse functions:

Polymorphism means that functions can be overloaded and used in many different situations. In such cases separating the profiling results of a function for each type of functionality is a difficult task. Often the programmer will just be faced with a single compound cost for a `member` function or a `read` function and, since splitting these costs between parent functions is difficult, it is hard to identify which usage of the function is responsible for consuming the most resources. Trying to split these compound costs using the GHC cost-centre profiler requires much time and effort; with the HBC heap profiler the task is nearly impossible.

Functional programs often use a number of recursive definitions. Monitoring and predicting the cost of a recursive function is simple. When functions become mutually recursive however, such a task becomes more complicated. Separating the costs of these functions, or more specifically identifying which of the mutually recursive functions is responsible for the large costs, is difficult.

Functional programs are composed of a large number of smaller expressions:

A functional program is often composed of a number of smaller functions. This can be a disadvantage when setting cost centres and trying to trace high profiling costs through programs. In many case studies it became a tiresome task, a large proportion of the effort being spent moving cost centres and recompiling the program to isolate parts of the code which were expensive. Once the problem code had been narrowed down most of the changes to the code itself were simple and required very little effort.

Moving cost centres in the code and recompiling is often carried out to narrow down the search for an expensive expression. Such a scheme is also used to identify why an expression is expensive, that is to identify the calling functions which

cause this. The cost-centre profiler allows a certain amount of flexibility in the gathering of costs as cost centres can be included around 'interesting expressions'. This method, though flexible, still requires a considerable amount of effort. The HBC heap profiler is not so flexible and does not allow individual expressions to be profiled. Inheriting costs is not therefore possible and the results are limited to low-level function definitions.

Profiling a program can take a long time:

The profiling task takes a long time. The LOLITA system takes a number of hours to compile and profiling slows this compilation process down quite considerably. For instance, current statistics show that compiling the system from scratch, using GHC with the profiling option, takes up to six times longer than without the profiling option. Of course it is rare in a functional programmed system that the whole program needs to be compiled from scratch; normally individual modules are recompiled and re-linked into the working system. However, switching between profiled and non-profiled versions requires a complete recompilation.

Once a profiled version of the system has been established, changes to the cost-centre annotations must be re-compiled into the executable system. Although this gives the programmer control over which parts of the program are profiled, it does mean that recompilation is required each time a change in the cost centres is made. The program must then be re-run and the new profiling results produced. It is conceivable that this process of changing the cost-centre annotations, recompiling and then re-running may need to be repeated a number of times before the expensive function (or functions) is identified. For this reason profiling a large program can quite possibly take a number of weeks!

Profiling overheads:

The GHC system of profiling is successful, although there are a number of large overheads which often make profiling difficult. The executable files are generally fifty percent larger than the un-profiled versions of the system. The amount of heap space needed to run the LOLITA system can be up to four times that of the un-profiled version (heap profiling with GHC 0.22); this can make large programs

almost un-profilable on some machines[5]. The execution time of the LOLITA system is also greatly increased when profiling the system. These overheads are generally around one hundred and fifty percent, although, due to the large heap sizes needed to run the code, paging can increase the overheads to hundreds of percent.

These overheads would be acceptable if they were indeed one-off costs. However, these costs are imposed each time the profiling is done. It is proposed that the profiling task should be a one-off collection of costs which can then be post-processed. It would then matter less if the collection of costs took five minutes extra than normal execution because the costs could quickly be manipulated later.

Potential profiling hazards:

CAFs — The GHC cost-centre profiler is particularly sensitive to Constant Applicative Forms, top-level values which are not functions. For example the expression '`numbers = [ 1, 2, 3, .. ]`' is an arbitrarily large CAF representing the infinite set of positive integers greater than or equal to 1. Such programming techniques are to be avoided as the GHC garbage collectors do not collect the heap reachable form of the CAF, leaving open the possibility of space leaks. Profiling is also obscured by the presence of CAFs. It is possible to trace some of the CAF costs, which can be responsible for up to 40% of run-time costs in LOLITA, but those identified as prelude CAFs are effectively untraceable, as it is not possible to profile the library and prelude modules of the system (GHC 0.22).

Choice of profiling scheme — There are advantages and disadvantages to using both automatic profiling (using the `-auto-all` compiler option) and user-annotated profiling (using the `-prof` compiler option) with the cost-centre profiler. The automatic profiling approach does avoid the problem of where to place a cost-centre expression but results are presented at a low level. Alternatively, complete control is given to the profiling process by the hand-annotation profiling method. The integrated use of both schemes brings about the most impressive benefits.

[5]Implementing the profiler with the generational garbage collector is expected to reduce this problem [Sansom, 1994].

4.6 Proposed Improvements to Profiling Tools

With the results of these case studies as a basis, a number of proposals are made to extend the field of profiling. These proposals are based on improvements to the profiling tools, which in themselves will add further support to the task of profiling.

The ability to profile time and heap space is a major advantage over just heap profiling. Cost-centre profiling is also more flexible than the HBC heap profiling scheme as the code to be profiled can be identified by the programmer. The proposals therefore take the form of improvements to cost-centre profiling.

The time in which profiling experiments can be performed could be greatly reduced by the introduction of a post-processing scheme which would allow results which had previously been collected from a profiled program to be analysed. Much of the time spent profiling with cost centres is needed because cost centres have to be re-positioned in the code and then the system has to be recompiled. A post-processing scheme is proposed so that compilation is only required once, even if cost centres are changed; the post-processor will simply manipulate the existing results accordingly.

Together with this post-processing scheme, an accurate method of cost inheritance is proposed. With off-line cost-processing more attention can be paid to the aggregation of costs in the call-graph. By modifying the current profiling scheme to provide more detailed information, cost inheritance can be combined within a post-processing environment. This will prevent users from having to move the cost centres each time they want to see how the cost information is distributed between functions. The collection of more detailed profiling information, based on stacks of functions, is described in the next chapter.

Profiling a program in terms of stacks of functions will produce more program information which should help the programmer to follow costs through the program correctly. Identifying costly calls to functions will not be based on intuition and the lazy evaluation behaviour of the program can be identified by particular stacks. This should speed up the identification of problems and prevent the programmer

from following any blind alleys in the identification of efficiency bugs.

The proposal of further profiling information may seem pointless in the light of current large profiling costs[6], but these schemes will be effective if an efficient implementation is considered. Particular attention is paid to this in the next chapter.

4.7 Conclusions

The study of profiling the LOLITA system over a two-year period has produced a number of unique case studies. The LOLITA system is developed by a variety of programmers who have varied programming experience. Many of the programmers work on applications based on the core of the LOLITA system; they are not expert functional programmers or experts on the detailed construction of the LOLITA system. Many of their experiences of profiling have therefore been unique tests of the current tools available; this applies with respect to the users of the tools and also the system on which these tools were tested.

The results from the series of profiling case studies performed on the LOLITA system were recorded. Particular attention was paid to the recording of criteria from which the case studies could be structured and re-constructed. The results of these case studies have been incorporated to provide a basis on which proposals can be made to improve the current profiling tools to make their use more effective in the LOLITA environment. It is hoped that the benefits which these changes bring about will also be useful to application and system programmers who program other large lazy functional systems.

The case studies presented are selected to demonstrate the use of the profiling tools in the maintenance of existing LOLITA code. They cannot show the time involved in the profiling of the LOLITA system, which is one of the prime motivations for the development of a post-processing tool. LOLITA is such a sizable system that any improvement to the analysis of its code, requiring less compilation

[6]though it is noted that the most problematic of these overheads is the re-compilation needed each time the cost centres are changed.

and modifications to the compile and run-time parameters, will be of enormous benefit to the development team.

The proposals suggested at the end of the chapter form the basis of changes made to the Glasgow Haskell Compiler which are documented in the remainder of this thesis. The implementation of these improvements and an extension to the current profiling theories are discussed in this context.

4.8 Chapter Summary

This chapter documents three of the case studies of profiling the LOLITA system. These were selected from a collection of case studies gathered over a period of two years.

The first of these case studies describes the results gained from profiling the complete system. Using the `-auto-all` compiler flag of the cost-centre profiler all globally-defined functions were profiled. The profiling results, using a balanced set of input data, highlighted those low-level functions which were heavily used. Improvements to these sections of code allowed overall improvements to be made to the execution time. This method of profiling quickly showed limitations as it did not demonstrate the relationship between low-level functions and their parent functions. Consequently it was not always clear why the use of a low-level function was expensive. An accurate method of cost inheritance would have improved the profiling results and may also have reduced the time needed to perform this case study.

The second case study was based on the heap profiling of the LOLITA grammar transformations. The transformations were failing because they quickly ran out of heap space. The signs were that there was a space leak though it was not certain where in the code this was. The case study demonstrates how, using the profiler, the part of the code causing the space leak was identified. Although the profiler indicated what was causing the large amount of heap, it did not identify which part of the code was responsible for retaining the space. It is proposed that the

retainer profiler currently being developed by Runciman and Röjemo would help solve such problems.

The final case study followed the continuing development of the grammar transformations. This also focussed on heap profiling. The heap profiling graphs again indicated what part of the code was responsible for a large increase in the heap usage, though once more it was not certain why this should be the case. A number of strictifying operations were performed on the code as it was thought that these parts of the code were causing the problem. These changes made no difference to the heap results. Experience of the system finally led to the identification of the problem, a function eight generations away from where the profiling results were displayed. The debugging of this part of the code took four days to complete. Much of the time spent was because of the recompilation needed each time cost centres were moved in the code. In response to this a method of inheriting results using a post-processor is proposed.

These case studies and the others performed lead to a number of proposed improvements to the cost-centre profiler. A post-processing scheme would reduce the amount of time spent moving costs centres in the code and recompiling the LOLITA system. An accurate inheritance scheme would reduce the ambiguity caused by the costs of shared functions. These two ideas combined should bring a number of improvements to the problems identified with the current profiling tools.

Chapter 5

Cost-Centre Profiling

5.1 Introduction

Profiling the LOLITA system has allowed a number of problems with the current profiling tools to be identified. From the identification of these problems can be proposed methods which may help in the profiling of large systems. In particular it is desirable to offer two new facilities:

- The possibility of fully accurate inheritance profiles.
- The possibility of obtaining new flat and inheritance profiles without recompiling or re-running the program. This will allow the user to select and deselect cost centres in the code to obtain new results, while impose no further compilation or execution overheads.

The first of these facilities will offer an alternative way of displaying the profiling results. The user may view the profiling costs as a flat profile or as an inherited profile. The inherited profile will aid the programmer in his search for inefficient parts of the program by displaying the propagated costs at a higher level in the function hierarchy. This will not only help to identify which part of the program is expensive, but also the particular order of function calls which might show why it is expensive.

The second of these facilities will allow the user to select and de-select parts of the code just as he can with the current cost-centre profiler, but without having to re-compile and re-run the program each time the cost centres are changed. Instead, a post-processing facility will allow the movement of cost centres, and thus the associated results, after a single execution of the program. The benefit here is the time gained by not having to re-compile and re-run the program. These time gains are particularly important with large systems such as LOLITA which can take a number of hours to compile.

A new design for a profiler is proposed based on an extension to the existing Glasgow cost-centre profiler. The cost-centre profiler was introduced in chapter 2. This chapter pays more detailed attention to the raw details of the construction of the cost-centre profiler, to its formal semantics and its integration with the Glasgow Haskell Compiler. These need to be understood before the proposed changes to the profiler and compiler can be documented.

5.2 Profiling with Cost Centres

When profiling, the programmer is required to identify the portion of the program which he is interested in analysing in terms of its run-time costs. This identification of parts of the program may be performed automatically [Runciman and Wakeling, 1992] or more explicitly by the programmer annotating the source-level code [Sansom and Peyton Jones, 1992]. It is argued that the latter approach gives the programmer more control over selecting the parts of the code in which he is interested, though automatic annotation can prove to be particularly useful for profiling large programs. Cost-centre profiling makes use of both these techniques and thus offers a balanced and flexible approach.

A cost centre is described as a label to which costs are assigned. During profiling the programmer may annotate the code with an `scc` expression (set cost centre) which is followed by the cost-centre name. For instance

```
function x = scc "costOfFunction" (map expensiveFunction x)
```

will assign to the cost centre `costOfFunction` the costs of the evaluation of the expression `(map expensiveFunction x)`.

In large programs such a scheme could become quite difficult, so to avoid the programmer annotating every function definition, he is also able to select all top-level functions, functions in a named module or just those explicitly added by hand.

5.2.1 Cost-attribution semantics for cost-centre profiling

Costs are allocated to a single cost centre which is currently in *scope* according to a set of cost-attribution rules. Such rules state that given an expression, "`scc` *cc exp*", the costs attributed to *cc* using a lexical profiling technique are the entire costs of evaluating the expression *exp* as far as the enclosing cost centre demands it, excluding, firstly the costs of evaluating the free variables of *exp*, and secondly the costs of evaluating any `scc` expressions within *exp* (or within any function called from *exp*).

This means that any costs incurred by functions within the scope of the enclosing cost centre are aggregated (except those which are themselves profiled) and that the lazy evaluation order of the expression is maintained.

The behaviour of cost aggregation is specified using cost semantics [Sansom, 1994] which are based upon Launchbury's natural semantics for lazy evaluation [Launchbury, 1993]. Since these cost semantics avoid any ambiguity that an informal description might introduce, they are employed in the description of the cost-centre profiler. A judgement form is written to describe the semantic state of evaluating an expression:

$$cc,\ \Gamma\ :\ e\ \Downarrow_\theta\ \Delta\ :\ z,\ cc_z$$

This reads that, in the context of heap Γ and the current cost centre *cc*, the

expression e evaluates to the value z, producing a new heap Δ and a new current cost centre cc_z.

The costs of this evaluation are recorded in θ, a set of mappings between cost centres and costs; $cc \mapsto A$, for instance, would represent the cost of an application charged to the cost centre cc. The operator $\uplus$ is used to tally up the total costs for cost centres.

Therefore,

$$\theta_1 \uplus \theta_2(cc) \;=\; \theta_1(cc) + \theta_2(cc),$$

where, if we define

$$\theta = \{cc_1 \mapsto n_1, ..., cc_k \mapsto n_k\},$$

then the following holds

$$\theta(cc) = n_i, \;\texttt{if}\; cc = cc_i \;\texttt{else}\; = 0.$$

Costs are recorded in the domain of the $cc \mapsto cost$ mappings with the following constants:

R_λ: is the cost of returning a lambda abstraction;

R_C: is the cost of returning a closure;

H: is the cost of allocating a closure in the heap;

V: is the cost of evaluating a variable;

U: is the cost of an update;

A: is the cost of a curried application;

C: is the overhead of a case expression;

P: is the cost of a primitive application.

Semantic rules for the evaluation of lambda and constructor expressions, application rules, variable let and case rules, are defined in Sansom's thesis [Sansom, 1994] and shown in Figure 5.1.

$$\frac{cc_{scc},\ \Gamma\ :\ e\ \Downarrow_{\theta}\ \Delta\ :\ z,\ cc_z}{cc,\ \Gamma\ :\ \texttt{scc}\ cc_{scc}\ e\ \Downarrow_{\theta}\ \Delta\ :\ z,\ cc_z}$$

SCC

$$cc,\ \Gamma\ :\lambda x.e\ \Downarrow_{\{cc\mapsto R_\lambda\}}\ \Gamma\ :\ \lambda x.e,\ cc$$

Lambda

$$\frac{cc,\ \Gamma\ :\ e\ \Downarrow_{\theta_1}\ \Delta\ :\ \lambda y.e',\ cc_\lambda \quad (cc\ \oplus\ cc_\lambda),\ \Delta\ :\ e'[x/y]\ \Downarrow_{\theta_2}\ \Theta\ :\ z,cc_z}{cc,\ \Gamma\ :\ e\ x\ \Downarrow_{\{(cc\ \oplus\ cc_\lambda)\mapsto A\}\uplus\theta_1\uplus\theta_2}\ \Theta\ :\ z,\ cc_z}$$

Application

$$\frac{SUB(cc_e,cc),\ \Gamma\ :\ e\ \Downarrow_{\theta}\ \Delta\ :\ z,\ cc_z}{cc,\ \{\Gamma,\ x\overset{cc_e}{\mapsto}e)\}\ :\ x\ \Downarrow_{\theta_{res}}\ \{\Delta,\ WHNF(e,\ x\overset{cc_e}{\mapsto}e,\ x\overset{cc_z}{\mapsto}z)\}\ :\ z,\ cc_z}$$

where $\theta_{res}\ =\ \{cc\mapsto V\}\uplus\{cc_z\mapsto WHNF(e,O,U)\}\uplus\theta$
$WHNF(\lambda x.e,n,u)=n \qquad SUB(\text{“SUB”},cc)=cc$
$WHNF(C\ x_1...x_n,n,u)=n \quad SUB(cc_e,cc)=cc_z$
$WHNF(e,n,u)=u$

Variable

$$\frac{cc,\ \{\Gamma,\ x_1\overset{cc}{\mapsto}e_1,...,x_n\overset{cc}{\mapsto}e_n\}\ :\ e\ \Downarrow_{\theta}\ \Delta\ :\ z,\ cc_z}{cc,\ \Gamma\ :\ \texttt{let}\ x_1{=}e1,...,x_n{=}e_n\ \texttt{in}\ e\ \Downarrow_{\{cc\mapsto n*H\}\uplus\theta}\ \Delta\ :\ z,\ cc_z}$$

Let

$$cc,\ \Gamma\ :C\ x_1...x_n\ \Downarrow_{\{cc\mapsto R_C\}}\ \Gamma\ :\ C\ x_1...x_n,\ cc$$

Constructor

$$\frac{cc,\ \Gamma\ :\ e\ \Downarrow_{\theta_1}\ \Delta\ :\ C_k\ x_1...x_{m_k},\ cc_C \quad cc,\ \Delta\ :\ e_k[x_i/y_i]\overset{m_k}{i=1}\ \Downarrow_{\theta_2}\ \Theta\ :\ z,\ cc_z}{cc,\ \Gamma\ :\ \texttt{case}\ e\ \texttt{of}\ \{C_i\ y_1...y_{m_i}\ \texttt{->}\ e_i\}\overset{n}{i=1}\ \Downarrow_{\{cc\mapsto C\}\uplus\theta_1\uplus\theta_2}\ \Theta\ :\ z,\ cc_z}$$

Case

$$\frac{cc,\ \Gamma\ :\ e_1\ \Downarrow_{\theta_1}\ \Delta\ :\ z_1,\ cc_1 \quad cc,\ \Delta\ :\ e_2\ \Downarrow_{\theta_2}\ \Theta\ :\ z_2,\ cc_2}{cc,\ \Gamma\ :\ e_1\ \oplus\ e_2\ \Downarrow_{\{cc\mapsto P\}\uplus\theta_1\uplus\theta_2}\ \Theta\ :\ z_1\ \oplus\ z_2,\ cc}$$

Primitive

Figure 5.1: Cost Augmented Semantic Rules for Haskell.

5.2.2 Reduction rules and reduction sequences

The *SCC* rule in Figure 5.1 evaluates the expression e to the new expression z in the context of the annotating cost centre cc_{scc} and the heap Γ. This rule ensures that the cost reported from the evaluation of expression e will respect the scope of the `scc` expression. The costs will be stored in the set of costs θ, although no costs are given for the reduction of the `scc` expression itself. The resulting heap is shown as Δ and the cost centre as cc_z.

The rules for Lambda and Constructor expressions simply reduce lambda abstractions and constructors to themselves within the context of an enclosing cost centre. The cost of the evaluation is attributed to the enclosing cost centre with the mapping $cc \mapsto R_x$, where x refers to either λ, the cost of returning a lambda abstraction, or C, the cost of returning a constructor.

Similar rules exist for applications. The term on the left is reduced to a λ-abstraction, the argument is substituted for the λ-variable and the reduction continues. Interestingly the distinction between evaluation and lexical scoping can be defined here. The cost of evaluating the body of the lambda expression can be attributed to cc, the cost centre enclosing the application, or cc_λ, the cost centre of the λ-abstraction.

The Variable rule describes a heap binding shown using the notation $x \mapsto (cc_e)\ e$. The cost of evaluating the bound expression e is attributed to the cost centre cc_e; this is the cost centre annotated by the declaration site. A distinction is made however if the cost centre is a `"SUB"` cost centre; that is, the cost centre is attached to a top-level function in Γ_{init}. Here the `"SUB"` is used to capture this choice. The components V and U refer to the cost of demanding the value of the variable and the cost of performing the update respectively.

The Let rule is simpler, extending the heap with the new bindings. The costs are calculated by multiplying the cost of allocating a closure, H, by the number of bindings, plus the costs of evaluating e itself.

The Case rule reduces the body and each arm of the case in the context of the reducing cost centre *cc*. The primitive rule evaluates each argument in the context of *cc* and applies the primitive operator $\oplus$. The cost of this application is also mapped to *cc*.

The semantic rules are defined as *premises*, the sequents above the line, and the *conclusion*, the sequent below the line. An instance of the application of a logical rule is called an *inference* if it is applied from the premise to the conclusion. Such an instance is called a *reduction* if it is applied in an inverted fashion from the conclusion to the premises. Now that these rules have been described, they can be used in reduction sequences for expressions written in Haskell. Reduction sequences of this nature are expressed as proof trees. Rather than showing them in tree form, Sansom chose to stress the sequential nature of the reduction by setting out the proofs vertically. So for example, $cc, \Gamma : e \Downarrow_\theta \Delta : z, cc_z$ is written:

1 cc, $\{\Gamma\}$: e
2^x a sub-proof
3^y another sub-proof
4^z $\{\Delta\}$: z, cc_z, θ

The notation in this thesis is adopted from the work by Launchbury and Sansom with subtle differences included to make the reading of the proof easier. Each reduction step is numbered so that parts of the proof can be referred to individually. The rules employed in the derivation are indicated by the exponent of each proof step, eg. 2^x indicates that the second step in the reduction is attained by applying the rule x.

Each derivation may be made up of many smaller derivations. This is demonstrated in lines 2 and 3. At any one point in the derivation a cost centre is currently in *scope* if the costs of the reduction are associated with that particular cost centre. As the reduction proceeds, the current cost centre in scope will change and thus different parts of the program have their evaluation attributed to different cost centres. In the example above, the cost centre *cc* is in scope for the derivation steps 2 and 3. This simple case is expanded upon.

An example is considered which specifically makes use of the SCC^1 rule. The *SCC* rule evaluates the expression e to the new expression z in the context of the annotating cost centre cc_{scc} and the heap Γ.

Working from the conclusion of the rule to the premises (reduction), this rule states that the current cost centre, *cc*, is replaced with the cost centre following the `scc` annotation. So for example, in the code:

```
scc "plus" 2 + 3
```

The reduction of the `scc` annotation sets the current cost centre to the new cost centre `plus`. The costs of evaluating the primitive + and the constructors 2 and 3 are attributed to the cost centre `plus`, as the expression is in the scope of `scc`. The proof of the judgement: *cc*, Γ: `scc "plus" 2 + 3` $\Downarrow_{\theta''}$ Δ:5, `"plus"`, follows the form:

1 current cost centre *cc*, Heap Γ: `scc "plus" 2 + 3`

2^{scc} `"plus"`, Γ:2 + 3

3^{prim} a sub-proof of the primitive + returning $\theta' = \theta \uplus \{plus \mapsto P(+)\}$

4^{const} a sub-proof of the constructors 2 and 3 returning $\theta'' = \theta' \uplus \{plus \mapsto (C(2) + C(3))\}$

5^{prim} Heap Δ:5, `"plus"` θ''

6^{scc} Δ:5, `"plus"` θ''

The resulting set θ'' of reduction costs is:

$$\{plus \mapsto P(+), plus \mapsto (C(2) + C(3))\}^2$$

A slightly more complicated expression:

```
scc "times" 2 * 46 * (scc "plus" 4 + 8 )
```

[1]This choice is not made arbitrarily. In fact it is this rule which is modified later in the thesis.

[2]Where P returns the costs incurred in reducing the primitive operator found as its parameter, and C returns the cost of reducing the constructor found as its parameter.

will follow the scoping rules shown in the following proof of the judgement:
cc, Γ: `scc "times" 2*46*(scc "plus" 4+8)` $\Downarrow_\theta$ Δ:1104, `"times"`:

1 *cc*, Heap Γ: `scc "times" 2 * 46 * (scc "plus" 4 + 8 )`

2^{scc} `"times"`, Γ: `2 * 46 * ( scc "plus" 4 + 8 )`

3^{prim} a sub-proof of the primitive $*$ returning $\theta^1 = \theta \uplus \{times \mapsto P(*)\}$

4^{const} a sub-proof of the constructors 2 and 46
returning $\theta^2 = \theta^1 \uplus \{times \mapsto (C(2) + C(46))\}$

5^{prim} Heap Δ : `92 * ( scc "plus" 4 + 8 )`, `"times"` θ^2

6^{prim} a sub-proof of the primitive $*$ returning $\theta^3 = \theta^2 \uplus \{times \mapsto P(*)\}$

7^{scc} `"plus"`, Γ : `92 * : 4 + 8`

8^{prim} a sub-proof of the primitive $+$
returning $\theta^4 = \theta^3 \uplus \{plus \mapsto P(+)\}$

9^{const} a sub-proof of the constructors 4 and 8
returning $\theta^5 = \theta^4 \uplus \{plus \mapsto (C(4) + C(8))\}$

10^{prim} Heap Δ : `92 * 12`, `"plus"` θ^5

11^{scc} Heap Δ:1104, `"times"` θ^5

12^{scc} Heap Δ:1104, `"times"` θ^5

The resulting set θ^5 of reduction costs is:

$$\{plus \mapsto (C(4) + C(8)), plus \mapsto P(+), times \mapsto P(*), times \mapsto (C(2) + C(46)), times \mapsto P(*)\}$$

These semantic rules describe a complete language within which any reduction of a Haskell expression can be demonstrated whilst describing exactly where the cost of the reduction should go.

These cost semantics are extended later on in the thesis when our new profiler is described.

5.2.3 Push-enter reduction semantics

Throughout the execution of a program a *current cost centre* register, the cost centre in the context of the expression currently being executed, is stored. This register allows the costs of an expression to be attributed to the correct cost centre during a program's execution. This register is also used to return the resulting cost centre cc_z with the result of an expression.

The idea of having a current cost register is common to most profiling schemes, though the method by which the current register is loaded and restored is of most interest. The GHC compiler has cost-centre profiling defined for lexical, evaluation and hybrid (combining the two) methods of profiling. These schemes offer very different techniques for the loading and storing of the current cost centre register. Much attention has been dedicated during the development of the cost-centre profiler to the actual value of the current cost centre at a particular point during the execution of a program.

The STG-machine implementation [Sansom and Peyton Jones, 1992], on which the Glasgow Haskell Compiler is based, relies on a *push-enter* model of reduction. Semantics for a push-enter model of reduction differ from the eval-apply method of reduction, shown in the semantics of section 5.2.1, since the treatment of function application is significantly different. The eval-apply model of reduction evaluates the λ-abstraction being applied and then evaluates the body of an expression. The push-enter model of reduction pushes the argument being applied on to an argument stack and then enters the function expression; when the evaluation of the function is complete the argument is retrieved from the stack and the body evaluated without returning the λ-abstraction.

This difference caused Sansom to define a further set of semantics, based on an abstract push-enter reduction model, to describe the implementation of the current cost centre register. These definitions can then be mapped directly onto the architecture of the STG-machine.

The difference between the push-enter semantics and the eval-apply semantics is characterised by the addition of the argument stack. The argument stack is an ordered sequence: () is used to represent the empty stack and $a : as$ is used to represent the stack obtained by pushing the argument a onto the top of the stack as. A judgement form for the push-enter semantics, with cost-centre information, is similarly defined:

$$cc,\ \Gamma\ as\ :\ e\ \Downarrow_\theta\ \Delta\ :\ z,\ cc_z$$

This is read: the term e in the context of the set of (annotated) bindings Γ, argument stack as and enclosing cost centre cc, reduces to the value z together with the (modified) set of (annotated) bindings Δ and the resulting cost centre cc_z, attributing reduction costs to θ.

During the course of the evaluation the argument stack will be consumed by the expression being evaluated and the heap may be extended with new bindings or may have old bindings updated with their results. λ-abstractions are only returned if there are no arguments available to apply; this is therefore where the two evaluation models differ, as in the eval-apply semantics there is no argument stack and a λ-abstraction is always returned to an application site before being applied.

The difference in the reduction rules between the eval-apply and the push-enter models is demonstrated in [Sansom, 1994]. The difference between the two models is simple and the changes which are implemented in the eval-apply semantics are simply translated to the push-enter semantics. For this reason these further semantics are not included.

5.3 Compilation by transformation

A number of compilers are characteristically defined to perform compilation by transformation. That is, as much of the compilation process as possible is expressed as correctness-preserving transformations, each of which transforms a program into a semantically-equivalent program which should ideally work more quickly or in less

space [Peyton Jones and Santos, 1994]. Functional languages are particularly responsive to this process because they have a rich family of possible transformations based on the principle of referential transparency.

The Glasgow Haskell Compiler is an optimising compiler. This means that during compilation transformations are performed on the code for it to reach an efficient executable form.

Compilation by transformation is autonomous; that is, it uses transformations which can safely be applied automatically by a compiler. This is a method derived from an alternative manual method of *programming by transformations*, which is supported by programming environments while being directed by an experienced programmer.

Automatic program transformations fall into two broad categories:

- *Global transformations* — also referred to as *Glamorous transformations*. Examples of these include lambda lifting, full laziness, closure conversion, deforestation, transformations based on strictness analysis and so on.
- *Local transformations* — also referred to as *Humble transformations*. Individually these may look trivial

```
let x = y in E[x]  ==>  E[y]

case (x:xs) of (y:ys)  ->  E[y,ys]  ==>  E1[x,xs]
               []      ->  E2
```

The notation `E[]` stands for an arbitrary expression with zero or more holes. The notation `E[e]` denotes `E[]` with the holes filled in by the expression `e`.

Although global transformations may seem naturally desirable, it is important to bear in mind that local transformations are simple, that there are many more of them to consider, and many more opportunities to apply them.

In the Glasgow Haskell Compiler all local transformations are performed by the *simplifier*.

Whilst recognising that these transformations make the compilation process more effective, it was necessary to consider how this mechanism would affect the scope of cost centres in the code. Sansom recognised that most of the transformations performed would not affect the cost centres in the code, and saw that it was also possible to transform code around the cost centres, provided that none of this code passed the cost-centre boundaries. It was only when these boundaries were affected that suitable rules which would preserve the cost-centre scoping would have to be considered. At worst these optimising transformations would simply be ignored, producing slightly less efficient code yet producing the profiling information which was true to the code.

5.3.1 The CORE language

The simplifier performs transformations on the Core language, see Figure 5.2. This is described as a desugared version of Haskell, that is a simpler language with, for example, list comprehensions and pattern matching already transformed into simple constructs. In more graphic terms the Core language is also described as being a variant of the second-order lambda calculus augmented with the constructs `let(rec)`, `case` and `scc`.

The concrete syntax in the Core language is conventional: parentheses are used to disambiguate; application associates to the left and binds more tightly than any other operator; the body of the lambda abstraction extends as far to the right as possible; the usual infix arithmetic operators are permitted; the usual syntax for lists is allowed, with the infix '`:`' and '`[]`'; and where the layout makes the meaning clear, semi-colons may be omitted between bindings and `case` alternatives.

Bindings in `let` expressions are all simple; the left hand side of the binding is always just a variable. Function bindings are expressed by binding a variable to a lambda abstraction. Similarly the patterns in a `case` expression are all simple;

Program	$prog \rightarrow binds$	
Bindings	$binds \rightarrow bind_1; ..; bind_{n\geq 1}$	
	$bind \rightarrow var = expr$	
Expression	$expr \rightarrow$ `let` $bind$ `in` $expr$	Local definition
	| `letrec` $binds$ `in` $expr$	Local recursion
	| `\` $var_1 ... var_{n\geq 1}$ `->` $expr$	Lambda abstraction
	| $expr\ atom$	Application
	| $\Lambda\ tyvar$ `->` $expr$	Type abstraction
	| $expr\ ty$	Type application
	| `case` $expr$ `of` $alts$	Case expression
	| $constr\ atom_1 ... atom_{n\geq 1}$	Saturated constructor
	| $prim\ atom_1 ... atom_{n\geq 1}$	Saturated built-in op
	| `scc` $cc\ expr$	Set cost centre
	| $literal\#$	Unboxed object
	| var	
Alternatives	$alts \rightarrow aalt_1; ...; aalt_{n\geq 1};\ [def]$	Algebraic alts
	| $palt_1; ...; palt_{n\geq 1};\ [def]$	Primitive alts
Algebraic alt	$aalt \rightarrow constr\ var_1 .. var_{n\geq 1}$ `->` $expr$	
Primitive alt	$palt \rightarrow literal\#$ `->` $expr$	
Default alt	$def \rightarrow var$ `->` $expr$	
Atom	$atom \rightarrow literal\# \mid var$	

Figure 5.2: Syntax of the Core language.

nested pattern matching has been compiled to nested case expressions.

Other important points about the Core language are that:

- it is based on the second order lambda calculus (i.e. with type abstractions and type applications), extended with the constructs `let(rec)`, `case` and, in the case of the profiler, `scc` constructs.
- arguments are atomic, i.e. variables or literals.
- it supports un-boxed types and un-boxed classes.
- constructors and primitives are always saturated.
- the Core language has direct operational reading: allocations are represented by `lets` and evaluation by `cases`.

The Core language is used to aid program transformation by making expressions explicit:

1. All application arguments are made atomic; this forces the creation of argument closures to be made explicit using `let`-bindings.
2. Boxing and un-boxing of values are made explicit; this enables many low-level transformations usually relegated to the code generator to be expressed as Core to Core transformations.

At the centre of the compiler are a set of local transformations which simplify Core expressions. Most of the optimising program transformations in the compiler are performed on the Core language. As well as these, there are some more specialised transformations aimed at particular optimisations:

1. `let`-bindings may be floated outwards to increase sharing or inwards to avoid unnecessary allocation.
2. The worker/wrapper transformations arrange for strict functions' arguments to be passed un-boxed.

3. Intermediate list data structures are eliminated using foldr/build deforestation.

`scc` constructs are dealt with at every pass in the compiler, and the program transformations have been carefully considered so that they do not distort the scoping of the `scc` expressions.

5.3.2 The Glasgow Haskell Compiler simplifier

The transformation of Core to Core definitions is performed by the compiler simplifier. This proceeds as follows:

1. Analysis of the program determines the way each value is used; this includes occurrence counts and strictness information. These metrics can then be used to determine whether particular transformations are satisfied and will identify the `let`-bindings to be inlined. The inlining transformation is simply:

   ```
   let x = R in B[x]  ==>  B[R]
   ```

   Inlining is more conventionally used to describe the instantiation of a function body at its call site, with arguments substituted for formal parameters. The Glasgow compiler treats this as a two-stage process: inlining followed by beta reduction, the latter of which is defined as:

   ```
   (\x  ->  E[x]) arg  ==>  E[arg]
   (/\a ->  E[a]) ty   ==>  E[ty]
   ```

   Working with a higher-order language means that not all the arguments may be available at every call site. By separating inlining from beta reduction, one process at a time can be concentrated on.

2. This information determines how the program is simplified based on a set of transformations, Figure 5.3.

Name	Before	After
Unfolding or Inlining	`let` v = E_v `in` E	$E\,[E_v/v]$
Case Elimination	`case` E_v `of` v `->` E	$E\,[E_v/v]$
β-reduction	$(\lambda v.E)\ x$	$E\,[x/v]$
Let to Unboxing Case	`let` v = E_v `in` E	`case` E_v `of` $C\ v_1...v_n$ `->` `let` v = $C\ v_1...v_n$ `in` E
Let to Case	`let` v = E_v `in` E	`case` E_v `of` v `->` E
Constructor Reuse I	`let` v = $C\ v_1...v_n$ `in` ... $C\ v_1...v_n$ ...	`let` v = $C\ v_1...v_n$ `in` ... v ...
Constructor Reuse II	`case` v `of` ... $C\ v_1...v_n$ `->` $C\ v_1...v_n$ ...	`case` v `of` ... $C\ v_1...v_n$ `->` ... v ... ...
Case of known constructor	`case` $C_i\ v_1...v_n$ `of` ... $C_i\ v_1...v_n$ `->` E_i ...	$E_i\,[v_1/v_{i1}...v_{in}/v_n]$
Let Floating from Let	`let` v = `let` w = E_w `in` E_v `in` E	`let` w = E_w `in let` v = E_v `in` E
Let floating from Case	`case` (`let` v = E_v `in` E) `of` ...	`let` v = E_v `in case` E `of` ...
Let floating from App	(`let` v = E_v `in` E) x	`let` v = E_v `in` $E\ x$
Case Floating from Let	`let` v = `case` E_c `of` alt_1 `->` E_1 ... alt_n `->` E_n `in` E	`case` E_c `of` alt_1 `->` `let` v = E_1 `in` E ... alt_n `->` `let` v = E_n `in` E
Case Floating from case (Case of Case)	`case` (`case` E_c `of` alt_{c1} `->` E_{c1} ... alt_{cm} `->` E_{cm}) `of` alt_1 `->` E_1 ... alt_n `->` E_n	`case` E_c `of` alt_{c1} `->` `case` E_{c1} `of` alt_1 `->` E_1 ... alt_n `->` E_n ... alt_{cm} `->` `case` E_{cm} `of` alt_1 `->` E_1 ... alt_n `->` E_n
Case Floating from App	(`case` E_c `of` alt_1 `->` E_1 ... alt_n `->` E_n) v	`case` E_c `of` alt_1 `->` $E_1\ v$ ... alt_n `->` $E_n\ v$

Figure 5.3: Local Transformations.

scc expressions impose a restriction on these transformations, as evaluation must not be moved from the scope of one cost centre to the next if the cost centres are to record accurate results. Most local transformations can be performed without affecting the sub-expressions within them. However, there are two situations in which the cost centres and scc annotations affect these local transformations:

- Unfolding and case elimination perform substitutions on the complete expression. This may move the evaluation into the scope of another scc annotation.
- The applicability of transformations which match patterns consisting of more than one language construct may be hindered by an intervening scc construct.

To reduce this problem during the development of the cost-centre profiler, fewer optimisations were applied in these two cases (depending on the placement of cost centres). This did not impose much of a restriction on most of the transformations, although it was noted that a large number of let-floating transformations were hindered.

Sensibly, rules were introduced by Sansom which would allow let-floating within the cost-centre context, thus preserving a large number of these compiler transformations.

5.3.3 Let floating transformations

There are both local and global let-floating transformations in the Glasgow compiler. The global transformations come in two forms:

Outward Floating — Floating let-bindings out of lambda abstractions to improve sharing (similar to the full laziness transformation).

Inward Floating — Floating let-bindings inwards to avoid allocating the binding unnecessarily.

So that these transformation would succeed in the cost-centre framework, two further transformations, $T1$ and $T2$, were added. These rules annotate the right hand side of a `let`-binding with an $\texttt{scc}_{sub}$ when it is floated past an `scc`:

$(T1)$ `scc` cc `let` $v = E_v$ in $E \quad \Rightarrow \quad$ `let` $v = (\texttt{scc}_{sub}\ cc\ E_v)$ in (`scc` cc E)

The first transformation, $T1$, floats an `scc` expression into a `let`-binding (or floats a `let` out of the `scc` expression). The cost of reducing the expression E remains within the scope of the cost centre `scc`. The costs of reducing the expression E_v must also be kept within the scope of the cost centre *cc*. This is done using the $\texttt{scc}_{sub}$ annotation to indicate that although this sub-expression has been moved into the scope of a different cost centre, the costs of evaluation must still be attributed to the cost centre *cc*. Evaluating a $\texttt{scc}_{sub}$ expression does not increment the count of the expression instances evaluated; this is only incremented when the original `scc` expression is entered.

Therefore the costs of evaluating both E and E_v are attributed to the cost centre *cc*. This is the behaviour that would be expected in these circumstances. There is a slight anomaly with this rule as the cost of the `let`-binding is no longer in the scope of the cost centre `scc`, though this movement of a single reduction to the scope of another cost centre is unlikely to affect the results much.

$(T2)$ `let` $v = E_v$ in (`scc` $cc\ E$) $\Rightarrow$ `scc` cc `let` $v = (\texttt{scc}_{sub}\ ecc\ E_v)$ in E

The second transformation, $T2$, floats a `let`-binding into an `scc` annotation. The `let`-binding and the expression E are in the scope of the cost centre *cc* on the right of the transformation rule. The right hand side of the transformation must also be annotated with a $\texttt{scc}_{sub}$ expression, enclosing the expression E_v with the enclosing cost centre (*ecc*). This prevents the costs of evaluating the expression E_v from being incorrectly attributed to the cost centre *cc*.

It is important to note that this second transformation can only be performed if the enclosing cost centre *ecc* is known. It is later discovered that this is not implemented in GHC, as there is no way in the Glasgow compiler of keeping track of the enclosing cost centre.

Further optimisations are possible on the resulting code to effectively tidy up the results of the let-floating transformations. For example the following transformation, $T3$, is applied by the simplifier to remove the redundant cost centres in the code.

($T3$) If E_v has an scc expression on it then $\mathsf{scc}_{sub}\ cc\ (\mathsf{scc}\ cc'\ E_v) \Rightarrow \mathsf{scc}\ cc'\ E_v$

This transformation preserves the immediately enclosing cost centre, but removes any of the other cost centres, as in the cost-centre profiler these are seen as redundant.

These let-floating transformations allow the compiler optimiser to operate successfully and with little hindrance from the cost-centre profiler. In adding these transformations the scoping of the cost centres is preserved during compiler optimisation, so that the results of the profiler are indeed accurate and the programmer is able to profile optimised code as required.

5.4 Chapter summary

The Glasgow cost-centre profiler forms the basis on which a new profiler is constructed. In order to discuss the development of this new profiling system, it is necessary to consider the implementation of the cost-centre profiler in more detail. This chapter introduced the details needed for the discussion of the new work, which is presented in the next chapter.

The formal semantics used to describe the behaviour of the cost-centre profiler have been presented. These semantics allow reasoning about the mapping between program costs and the cost centres in the code. Some example reduction sequences were demonstrated.

The Glasgow Haskell Compiler is an optimising compiler. Optimisations are performed during compilation by a sequence of transformations to the code to produce effective executable code. It is important for the profiler writer to consider these transformations carefully so that expressions are not moved from the scope of

one cost centre into the scope of another. If this were to happen the profile results would be inaccurate.

Those transformations which would move program costs past cost-centre boundaries were considered. The simplest way to prevent these transformations from altering the profiling costs was to remove the transformations from the compiler. This had a limited effect on the efficiency of the program in all cases except the `let`-floating transformations.

These `let`-floating transformations were redefined by Sansom so that they preserved the cost-centre-scoping rules. The new `let`-floating transformations were discussed.

Chapter 6

Cost-Centre-Stack Profiling

6.1 Introduction

The method of profiling with cost centres discussed in the previous chapter is extended to include the notion of cost-centre stacks. These cost-centre stacks will form the basis of a new profiling scheme which will allow accurate cost inheritance and also profiling results which are amenable to post-processing.

The notion of a cost-centre stack as opposed to a cost centre is defined as follows. When program costs are recorded by the profiler they are attributed to the cost-centre stack which is currently in scope. The cost-centre stack is composed of the current cost centre and all those cost centres through which this cost centre was reached. This provides not only a record of what part of the program is current, but also the path that the program took in reaching this part of the program. The profiling results then take the form of many cost-centre stacks for which methods of cost inheritance and post-processing are defined.

Drawbacks with this method are contingent. The idea of recording this kind of information during the execution of a program is potentially disastrous, as the overheads may be huge and the quantity of results overwhelming. An important component of the work is therefore the way in which the profiler is implemented and integrated with the existing cost-centre profiler and Glasgow Haskell Compiler.

This chapter is composed of the following parts:

- The cost-centre profiling technique formally defined in the previous chapter is extended to include the notion of cost-centre stacks. The cost propagation techniques needed to implement cost inheritance are also formally defined in this chapter.
- The cost-centre-stack profiling scheme relies heavily on the efficiency of its implementation. It will only be effective if the results can be collected in a reasonable amount of time and the heap needed to execute the program is practical. The detailed implementation of cost stacks is discussed and examples are introduced.
- The Glasgow Haskell Compiler, on which the cost-centre-stack profiler is defined, is a complex system and inevitably the changes involved to include the new profiling scheme are detailed. The complexities of retaining compiler optimisations are discussed, enabling the cost-centre-stack profiler to be implemented and used on programs compiled with the GHC optimiser.
- Finally the post-processor is described. The display of the enhanced results is discussed and the method of inheritance and cost manipulation is introduced.

6.2 Cost-Centre Stacks

The current cost-centre system is extended to include the notion of a cost-centre stack [Morgan and Jarvis, 1995]. The objective of the cost-centre stacks is to record not just the immediately enclosing cost centre but all the enclosing cost centres to a certain part of a program.

Many of the failings of effectively producing inheritance profiles have been caused by the schemes used to inherit costs. The statistical inheritance method, for instance, does not account for calls to functions with very different arguments and, in order to produce more accurate results, it is necessary to record all the

enclosing cost centres to an expression; in this way costs can be unambiguously assigned to those parts of the program which cause them.

This scheme of recording all of the enclosing cost centres to an expression is created using cost-centre stacks. An earlier example (first seen in chapter 2) is used. Consider each function in the example to have a cost centre attached of the same name.

"The cost of a shared function h would be split between its calling functions, f and g, depending on how many calls were made to the shared function. For example if there are 8 calls from f to h and only 2 calls from g to h then the time spent in (or below) h will be divided 8:2".

Realistically these results might not be true: g may have called h twice with a list argument of size 100,000; f may have called h eight times with a list argument of size 10. By storing the function calls of h from g and h from f, it is possible to assign the costs of h to the two pairs (h, f) and (h, g); the programmer then knows exactly which pair caused the most costs and his attention is drawn to the correct part of the code.

It is possible to extend such a scheme to the parent function of the parent function and so on, until the results are in fact an accurate collection of all the results in the program's execution. For instance the above example may produce a set of results containing the stacks, $(h, f, main)$, $(h, g, main)$, $(f, main)$ etc., each with their own associated costs. The question as to how a particularly expensive function call came about is made easier, the programmer can investigate what the functions arguments were, and from where they originated. Clearly some scheme must be adopted to display such results; this is achieved by using post-processing which can be implemented so that the results are displayed either as a standard flat profile, or as an accurate inheritance profile. Variations to these schemes are possible; the important thing to recognise is that a complete set of results is now available which can be manipulated after the execution of the program.

There are a number of design criteria ($\mathcal{A}1$—$\mathcal{A}6$) which are drawn up for this profiling scheme:

$\mathcal{A}1$. That during *compilation* and program *execution* all top-level functions are deemed to be cost centres. The phrase "top-level functions" comes from [Sansom, 1994], it means lexically defined top-level functions, for example

```
functionA x = functionB (functionC x)
              where functionC = hd x

functionB x = x ++ x
```

`functionA` and `functionB` are top-level functions; `functionC` is not. This applies to any level in the program structure. All program costs will be recorded with regard to these top-level functions.

$\mathcal{A}2$. During *post-processing*, the stage at which profiling results are manipulated, cost centres may be selected or not by the programmer. Those cost centres which are not selected are called subsumed cost centres.

$\mathcal{A}3$. Profiling, at *execution* time, uses a cost stack rather than a single cost centre. Profiling results are therefore produced in terms of cost-centre stacks as well as cost centres. The cost stacks are used as input to the post-processor.

$\mathcal{A}4$. A cost stack is a sequence of cost centres ordered in such a way that its costs are attributed to the uppermost selected cost centre in the stack. For instance costs are attributed to the function *thirdFunction* in the cost stack $\langle thirdFunction, secondFunction, firstFunction \rangle$, as this is currently at the top of the cost-centre stack.

$\mathcal{A}5$. When entering a new cost centre at *execution* time, it should be pushed onto the stack. Any previous entry of the same cost centre is removed from the stack. This proposal is prompted by the need for an efficient implementation which is discussed in section 6.3. The resulting cost-centre stack replaces the current cost-centre stack.

$\mathcal{A}6$. When producing a profile during *post-processing*, the user is given a choice as to which cost centres he wishes to select and also whether he requires results

to be inherited or not ($\mathcal{A}2$). When inheriting costs the system locates the top-selected cost centre on each stack and adds the costs of the subsumed cost centres on the stack to the costs for that selected cost centre. This process is described in more detail in a later section.

6.2.1 Cost-attribution semantics for cost-centre stacks

The cost-centre stacks are introduced as an extension to the semantic rules; rather than modeling a single cost centre, cc_{scc}, a cost-centre stack is proposed ($\mathcal{A}3$).

Sequence notation, $\langle x_n, x_{n-1}, .., x_1 \rangle$, is used to represent a stack. Catenation of a cost centre, x_4, to a cost-centre stack, $\langle x_3, x_2, x_1 \rangle$, is performed using $x_4 \frown \langle x_3, x_2, x_1 \rangle$, and results in the sequence $\langle x_4, x_3, x_2, x_1 \rangle$ [Spivey, 1989]. The cost centre x_4 is said to be at the top of the stack; this corresponds to the current cost centre of Sansom's cost-centre profiler.

The cost-centre-stack extension is modelled in the semantic rules by modifying the reduction rule for the `scc` annotations. It is enough to modify this single rule and use the remainder of the rules in their current form.

The *SCC* rule states that to evaluate `scc` cc_{scc} e, we evaluate the expression e to the new expression z in the context of the annotating cost centre cc_{scc} and the heap Γ. The cost reported from the reduction of e will respect the scope of the `scc` expression and will be stored in the set of costs θ (no costs are given for the reduction of the `scc` expression.) The resulting heap is shown as Δ and the cost centre cc_z.

The original *SCC* rule appeared as:

$$\frac{cc_{scc},\ \Gamma\ :\ e \Downarrow_\theta \Delta\ :\ z,\ cc_z}{cc,\ \Gamma\ :\ \texttt{scc}\ cc_{scc}\ e \Downarrow_\theta \Delta\ :\ z,\ cc_z} \qquad SCC$$

To create a cost-centre stack the *SCC* rule is modified:

$$\frac{cc_{scc} \frown cc,\ \Gamma\ :\ e \Downarrow_\theta \Delta\ :\ z,\ cc_z}{cc,\ \Gamma\ :\ \texttt{scc}\ cc_{scc}\ e \Downarrow_\theta \Delta\ :\ z,\ cc_z} \qquad SCC'$$

This allows the current cost centre to be pushed onto the top of the cost-centre stack. The following proof shows how the example shown in the previous chapter then scopes with the new *SCC* rule.

1 *cc*, Heap Γ: `scc "times" 2 * 46 * (scc "plus" 4 + 8 )`

$2^{scc'}$ $\langle times \rangle$, Γ: `2 * 46 * ( scc "plus" 4 + 8 )`

3^{prim} a sub-proof of the primitive $*$ returning $\theta' = \theta \uplus \{\langle times \rangle \mapsto P(*)\}$

4^{const} a sub-proof of the constructors 2 and 46

returning $\theta^2 = \theta' \uplus \{\langle times \rangle \mapsto (C(2) + C(46))\}$

5^{prim} Heap Δ : `92 * ( scc "plus" 4 + 8 )`, $\langle times \rangle$ θ^2

6^{prim} a sub-proof of the primitive $*$

returning $\theta^3 = \theta^2 \uplus \{\langle times \rangle \mapsto P(*)\}$

$7^{scc'}$ $\langle plus, times \rangle$, Γ : $92* : 4 + 8$

8^{prim} a sub-proof of the primitive $+$

returning $\theta^4 = \theta^3 \uplus \{\langle plus, times \rangle \mapsto P(+)\}$

9^{const} a sub-proof of the constructors 4 and 8

returning $\theta^5 = \theta^4 \uplus \{\langle plus, times \rangle \mapsto (C(4) + C(8))\}$

10^{prim} Heap Δ : `92 * 12` $\langle plus, times \rangle$ θ^5

$11^{scc'}$ Heap Δ : `1104`, $\langle times \rangle$ θ^5

$12^{scc'}$ Heap Δ : `1104`, $\langle times \rangle$ θ^5

The results of a full example of the semantic reduction and inference, using both the original cost centres and the cost-centre stacks, are considered for the function definition:

```
f = scc "first" 1 + ( scc "second" 2 + 3 )
```

The first proof, which uses the original cost centres and cost-centre semantics:

1 INIT, Γ : `f = scc "first" 1 + (scc "second" 2 + 3 )`

2^{scc} `"first"`, Γ : `1 + (scc "second" 2 + 3 )`

3^{prim} `"first"`, Γ : `1`

4^{const} Δ : 1, `"first"` with $\theta = M'$

5^{prim} `"first"`, Γ : `(scc "second" 2 + 3 )`

6^{scc} `"second"`, Γ : `2 + 3`

7^{prim} `"second"`, Γ : `2`

8^{const} Δ : 2, `"second"` with $\theta = \{second \mapsto C(2)\}$

9^{prim} `"second"`, Γ : `3`

10^{const} Δ : 3, `"second"` with $\theta = \{second \mapsto C(3)\}$

11^{prim} Δ : 5, `"second"` with $\theta = M$

12^{scc} Δ : 5, `"second"` with $\theta = M$

13^{prim} Δ : 6, `"first"` with $\theta = M''$

14^{scc} Δ : 6, `"first"` with $\theta = M''$

where:

$$M = \{second \mapsto C(2)\} \uplus \{second \mapsto C(3)\} \uplus \{second \mapsto P(+)\}$$
$$M' = \{first \mapsto C(1)\}$$
$$M'' = \{first \mapsto P(+)\} \uplus M' \uplus M$$

gives the resulting set θ of associated costs:

$$\{first \mapsto (P(+) + C(1)),\ second \mapsto (P(+) + C(2) + C(3))\}$$

Where P returns the costs incurred in reducing the primitive operator found as its parameter, and C returns the cost of reducing the constructor found as a parameter.

The proof using the extended semantics created for the cost-centre stacks:

1 $\langle$INIT$\rangle, \Gamma$: `f = scc "first" 1 + (scc "second" 2 + 3 )`

$2^{scc'}$ S', Γ : `1 + (scc "second" 2 + 3 )`

3^{prim} $S', \Gamma : 1$

4^{const} $\Delta : 1, S'$ with $\theta = M'$

5^{prim} S', Γ : `(scc "second" 2 + 3 )`

$6^{scc'}$ $\langle second \rangle^\frown S', \Gamma : 2 + 3$

7^{prim} $\langle second \rangle^\frown S', \Gamma : 2$

8^{const} $\Delta : 2, \langle second \rangle^\frown S'$ with $\theta = \{\langle second \rangle^\frown S' \mapsto C(2)\}$

9^{prim} $\langle second \rangle^\frown S', \Gamma : 3$

10^{const} $\Delta : 3, \langle second \rangle^\frown S'$ with $\theta = \{\langle second \rangle^\frown S' \mapsto C(3)\}$

11^{prim} $\Delta : 5, \langle second \rangle^\frown S'$ with $\theta = M$

$12^{scc'}$ $\Delta : 5, \langle second \rangle^\frown S'$ with $\theta = M$

13^{prim} $\Delta : 6, S'$ with $\theta = M''$

$14^{scc'}$ $\Delta : 6, S'$ with $\theta = M''$

where:

$$
\begin{aligned}
S' &= \langle first, \texttt{INIT} \rangle \\
M &= \{\langle second \rangle^\frown S' \mapsto C(2)\} \uplus \{\langle second \rangle^\frown S' \mapsto C(3)\} \uplus \\
&\quad \{\langle second \rangle^\frown S' \mapsto P(+)\} \\
M' &= \{S' \mapsto C(1)\} \\
M'' &= \{S' \mapsto P(+)\} \uplus M' \uplus M
\end{aligned}
$$

returns the set θ of associated costs:

$$\{\langle first, \texttt{INIT} \rangle \mapsto (P(+) + C(1)),\ \langle second, first, \texttt{INIT} \rangle \mapsto (P(+) + C(2) + C(3))\}$$

Where INIT is used to indicate the initial state of the system, this will have no effect on the results themselves. By extending the existing cost semantics to produce cost centres as sequences, it is now possible to develop a theory of cost inheritance, which is later implemented in the profiling post-processor. Secondary semantics are also considered for the cost-inheritance theory.

6.2.2 Secondary semantics for cost inheritance

The resulting set of cost-centre stacks can be used to output the results in post-process form. A cost-centre stack contains cost centres which will be both selected by the user as 'interesting functions' and those cost centres for which results should simply be subsumed; these are referred to as *selected* and *non-selected* functions respectively. The selected functions will be stored in a set $SELECTED$.

The inheritance semantics are initially defined to deal with two types of post-processing, the first producing its output as non-inherited costs (a flat profile), and the second producing its output as inherited costs. The choice between these schemes is made by the user, $\mathcal{A}2$.

Firstly the non-inherited semantics. Each cost stack in the set θ will contain a top-selected cost centre, defined as the selected cost centre nearest the top of the stack. The costs associated with that particular cost stack are attributed to that top-selected cost centre. Consider for instance the stack $\langle c, b, a\rangle$ and the associated costs, say 10. The cost 10 is added to the selected cost centre c. The cost centre c will eventually contain all the costs for the stacks on which c is the top-selected cost centre; it is then printed as a post-processed result.

As an example consider the set θ of

$$\{\langle b, a\rangle \mapsto 10, \langle a\rangle \mapsto 20, \langle c, a\rangle \mapsto 10, \langle c, b, a\rangle \mapsto 50\}.$$

Under this scheme the non-inherited post-processed results,

$$a = 20, b = 10, c = 60,$$

would be produced. If the cost centre b was not selected, then the results

$$a = 30, c = 60,$$

would be printed instead. Such a scheme is defined more formally as:

$$\forall\, cc : cost\ centre;\ \theta : (seq\ cost\ centre) \rightsquigarrow \mathbf{N} \bullet$$

$$COST\ cc\ =\ sum\ \{\ cost \mid \forall\, S,\ T : seq\ cost\ centre;\ \exists\ stack : seq\ cost\ centre;\ cost : \mathbf{N} \bullet$$

$$(\{stack \mapsto cost\} \in \theta)\ \wedge$$

$$(S^\frown\langle cc\rangle^\frown T\ =\ stack)\ \wedge$$

$$(cc\ \in\ SELECTED)\ \wedge$$

$$(\forall\, cc' : cost\ centre \bullet cc'\ in\ S\ \Rightarrow\ cc' \notin SELECTED)\ \}$$

The operator $\rightsquigarrow$ represents an injective function and $\mathbf{N}$ represents the set of natural numbers.

It is possible to prove that using this scheme produces the same results as a flat cost-centre profile. All cost centres which are included in the original flat cost-centre profile are included in the set *SELECTED* for the cost-centre-stack profile.

The inherited costs are a simple variation on the above. Rather than adding the cost associated with a cost stack to the top-selected cost centre, the cost is added to all the selected cost centres within the cost stack. The results

$$a = 90, b = 60, c = 60,$$

are produced when the above example is considered under this inheritance scheme.

Again a formal definition of the scheme is offered:

$$\forall\, cc : cost\ centre;\ \theta : (seq\ cost\ centre) \rightsquigarrow \mathbf{N} \bullet$$

$$INHERITED\_COST\ cc\ =\ sum\ \{\ cost \mid \forall\, S,\ T : seq\ cost\ centre;\ \exists\ stack :$$

$$seq\ cost\ centre;\ cost : \mathbf{N} \bullet$$

$$(\{stack \mapsto cost\} \in \theta)\ \wedge$$

$$(S^\frown\langle cc\rangle^\frown T\ =\ stack)\ \wedge$$

$$(cc\ \in\ SELECTED)\}$$

This ensures that all selected cost centres have their costs inherited.

6.3 An Efficient Implementation

Profiling a program written using thousands of functions is potentially an expensive business. The cost-centre information recorded for each function will increase the execution time, which in turn may cause large programs to be almost un-executable.

Introducing cost-centre stacks may therefore seem a limited exercise, as cost stacks require more information to be stored than the original cost-centre model. An efficient implementation must therefore be considered to make such a scheme feasible.

Two efficiency problems were considered key in the implementation of cost-centre stacks; the first was the space needed to store these stacks during the execution of the program, the second was the time needed to build these cost-centre stacks and to access and add the profiling data to the stacks.

Space saving mechanisms

Cost-Centre-Stack Codes:

To produce a practical implementation, applicable to large as well as small programs, cost-centre stacks are replaced with *Cost-Centre-Stack Codes*; each cost-centre-stack code is derived from a *Cost-Centre-Stack Table.* The cost-centre-stack code acts as a pointer to the relevant stack entry in the cost stack table, so at any one time the cost-centre stack is simply a code which addresses an entry in the cost-stack table. See Figure 6.1.

The top left of the figure shows a reference to a cost-centre-stack code at an arbitrary point in a program's execution. This code, 0003, refers to the current cost-centre stack. This code references a point in the cost-stack table, the larger box below. Inside the cost-stack table (bottom right) a reference to the code 0003 is found; this is the representation of a stack. On the top of this stack is the cost centre B. A back pointer shows the reference to the cost-centre stack onto which B was pushed; this was the stack $\langle MAIN \rangle$. Therefore, the stack referred to by the

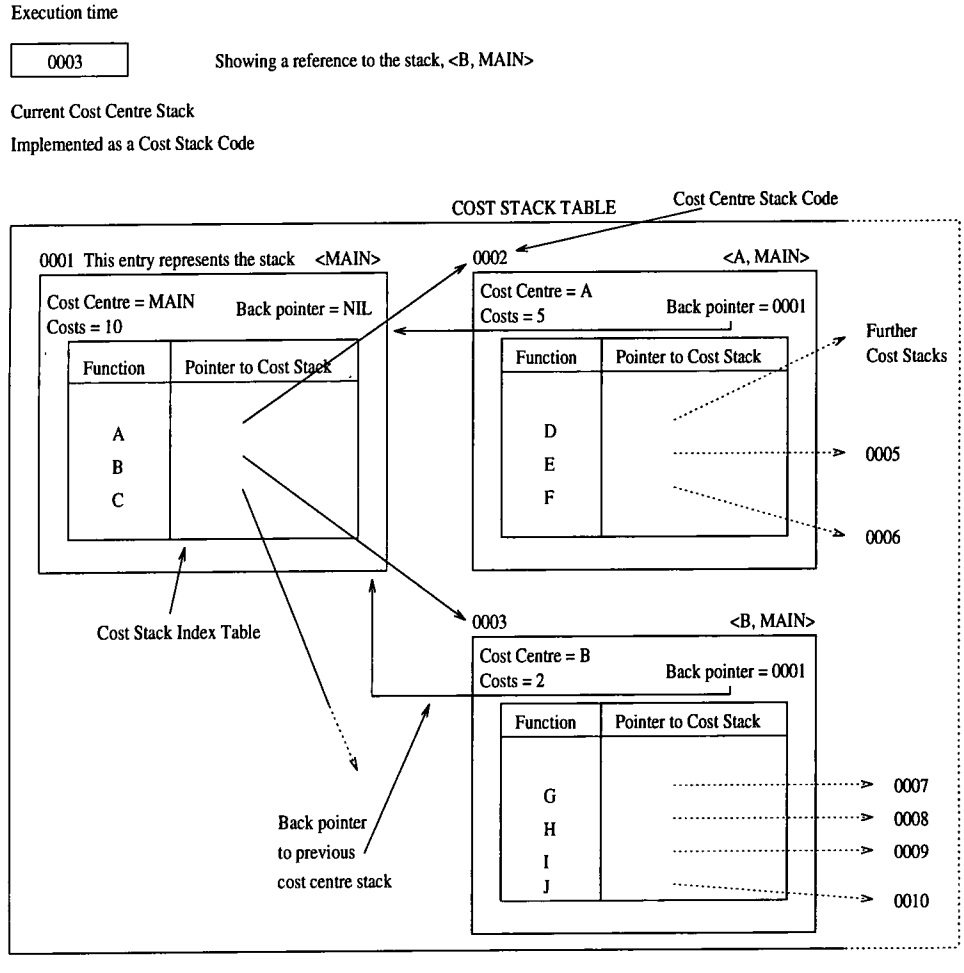


Figure 6.1: Implementation of Cost Stacks.

cost-centre-stack code 0003 is $\langle B, MAIN \rangle$.

In the cost-centre-stack table references to other cost-centre-stack codes are found. As an example it can be seen what cost-centre stacks these codes represent; the cost-centre-stack code 0001 represents the cost-centre stack $\langle MAIN \rangle$, 0002 represents the cost-centre stack $\langle A, MAIN \rangle$, 0006 represents the cost-centre stack $\langle F, A, MAIN \rangle$ and so on.

Compressed Stacks:

The cost-centre stacks themselves can be compressed. Consider the simple case of a function a which calls function b, this latter function then calls function a again[1]. To avoid this leading to an arbitrarily large number of stacks of the form $\langle a, b, a, b, .. \rangle$, only the top instance of a function call can be stored in a cost-centre stack (previously mentioned in $\mathcal{A}5$).

When a function is pushed onto the cost-centre stack, a previous instance of the function is removed from the stack. Such a scheme allows an infinite number of stacks to be stored in what is actually a very small, and finite, number of stacks. The example using two functions a and b requires two such stacks, $\langle a, b \rangle$ and $\langle b, a \rangle$, to represent an infinite number of stacks. Such stacks are referred to as *Compressed Stacks.*

This would seem a good mechanism for keeping the sizes of the stacks small. However, what would be the effect on the results themselves? Some figures are given for the example above to illustrate the behaviour of this model.

During the duration of the stack $\langle a \rangle$ 3 units of time are accrued. This is written as $\{\langle a \rangle \mapsto 3\}$, to represent the cost-centre stack and the costs attributed. When the function b is called, the cost-centre stack $\langle b, a \rangle$ is produced. During this part of the program 7 units of time are accrued. Finally the function a is called once again producing the compressed stack $\langle a, b \rangle$ and a further 1 unit of time is accrued.

[1]In the case of this example, and the others which follow, it is assumed that all top-level functions have a cost centre associated with them.

This produces the set of cost-centre stacks:

$$\{\langle a\rangle \mapsto 3, \langle b, a\rangle \mapsto 7, \langle a, b\rangle \mapsto 1\}$$

Using the secondary semantics for cost inheritance defined earlier $COST(a)$ produces the result 4 and $COST(b)$ produces the result 7; these are the non-inherited costs for each of these cost centres. $INHERITED\_COST(a)$ produces the inherited result for cost centre a, which is 11; $INHERITED\_COST(b)$ produces the result 8.

These results can be compared with the results produced by using uncompressed cost-centre stacks. The function calls above would produce the following set of uncompressed stacks:

$$\{\langle a\rangle \mapsto 3, \langle b, a\rangle \mapsto 7, \langle a, b, a\rangle \mapsto 1\}$$

The difference between the two models is determined by the cost-centre stack $\langle a, b, a\rangle \mapsto 1$. The non-inherited results are the same for this uncompressed stack as they are for the compressed stack; $COST(a)$ is 4, $COST(b)$ is 7. For the inherited results however the results are $INHERITED\_COST(a)$ is 12 and $INHERITED\_COST(b)$ is 8. It will be noticed that the inherited costs for a are 1 greater than before. The reason for this is that in the cost-centre stack $\langle a, b, a\rangle \mapsto 1$ the inheritance function adds 1 to cost centre a twice.

Is this really what is required? Should costs be added more than once to a cost centre in the inheritance? If a were in the cost-centre stack 50 times then these inherited results would become 61. This is certainly not desirable.

Using compressed stacks allows mutually recursive functions to be modelled with a fixed number of stacks. It also allows costs to recursive functions to be inherited once rather than a number of times, producing the inheritance results which are representative of the program's actual costs. The definition of cost inheritance therefore works precisely because of the fact that compressed stacks are used. If uncompressed stacks were used then the definition of inheritance would have to be re-written to prevent the multiple addition of a single cost.

Time issues

The second problem was that of time. Using the cost-centre-stack codes, shown in Figure 6.1, movement between cost-centre stacks can be achieved by simply referencing the centre-stack code to locate the next cost-centre stack. The expensive operation is in the building of new stacks, though, as the above example shows, this will occur relatively infrequently. There may be many thousands of calls between functions a and b, but once the two stacks representing them have been built they need only be referenced on any further calls by their cost-stack codes, so the only cost incurred here is by the look-up in the cost-stack table.

Each cost-centre stack stores profiling data representing the amount of the program's execution time which has been spent in that part of the program. Other profiling statistics can also be stored in the cost-centre stack such as heap profiling information and occurrence counts, but for the time being the discussion concentrates on time profiling. The time profiling information is continually updated during the execution of the program. The updating of these statistics in the cost-centre stacks as well as in the cost centres is not thought to cause any significant increase in the amount of time spent profiling.

6.3.1 The Push operation

The *Push* operation is written *Push(cost centre, cost-stack code)*. When performing a *Push*, the cost-stack code is used to reference the relevant entry in the cost-stack table. The table entry will contain the name of the top cost centre on the stack, a pointer to the previous cost-centre-stack table entry, and a *Cost-Centre Index Table.*

Notation is adopted to describe this:

X^n	Represents a cost-centre-stack table entry n with X as the cost centre at the head of the stack.
$X^n \Rightarrow Y^m$	Indicates the cost-centre-stack index table entry from cost-centre stack n to cost-centre stack m, given a push of Y onto n.
$X^n \leftarrow Y^m$	Indicates a back pointer from cost-centre stack m to cost-centre stack n.

The cost-centre stack is represented simply as the head of the stack, as the remainder of the stack can be accessed by the back pointer. Effectively, calls to *Push* are memoised.

There are a number of cases which need to be considered in the construction of cost centre stacks. Each individual case will help to demonstrate how the choice of structure is key in the implementation of efficient stacks and how neatly the scheme can be constructed.

Consider the cost-centre-stack table represented by the following:

$$\begin{array}{c} MAIN^1 \leftrightarrows A^2 \leftrightarrows B^3 \\ \Downarrow\uparrow \\ C^4 \end{array}$$

The cost-centre stack which contains the cost centre $MAIN$ is represented by the cost-centre-stack code 1; the cost-centre stack referred to by the code 4 is, $\langle C, A, MAIN \rangle$; the back pointer from this stack points to the cost-centre stack $\langle A, MAIN \rangle$.

Some examples of the push operation using this cost-centre-stack table can now be considered in turn.

Case 1

It has previously been mentioned that cost stacks are memoised. That is, once they have been created they continue to exist throughout the life of the program.

The first case considers reinstating a cost centre which already exists. For instance, in the cost-centre-stack table seen previously,

$$\begin{array}{c} MAIN^1 \leftrightarrows A^2 \leftrightarrows B^3 \\ \Downarrow\uparrow \\ C^4 \end{array}$$

pushing the cost centre B onto the cost stack represented by the cost-stack code 2, results in the cost-stack code 3.

This is a nice neat case since the stack which is required already exists. The overheads involved in such a case are of $O(n)$ for time and space, where n is the number of memoised entries contained in the cost-stack index table of cost-centre stack 2; this is typically between 1 and 20 (see chapter 7).

Case 2

In the second case the cost-stack index table would have no reference to that particular cost centre (there will have been no previous attempt to push this cost centre onto the current cost stack) and the cost centre would not have appeared anywhere in the cost stack. In this situation it is enough to add the new cost centre to the index table of the current stack table entry and thus create a reference to a new cost stack.

In the previous cost-centre-stack table, pushing the cost centre C onto the cost-centre stack with the code number 3 results in the following cost-centre-stack table and the cost-centre-stack code 5.

$$\begin{array}{c} MAIN^1 \leftrightarrows A^2 \leftrightarrows B^3 \leftrightarrows C^5 \\ \Downarrow\uparrow \\ C^4 \end{array}$$

The back pointers are used during this case to check to see if the cost centre C already exists in the cost-centre stack 3. The overheads of this case are therefore $O(n+m)$, where n is the number of memoised entries contained in the cost-centre-

stack index table of stack 3 and m is the depth of the cost-centre stack.

Case 3

The third case considers a cost centre already existing in a current stack.

First subcase:

The first subcase which is considered demonstrates that stacks can be built without any intermediate stacks having to be created.

Consider the case where the cost centre B is pushed onto the cost-centre stack identified by the code 5 in the following cost-centre-stack table.

$$\begin{array}{c} MAIN^1 \leftrightarrows A^2 \leftrightarrows B^3 \leftrightarrows C^5 \\ \Downarrow\uparrow \\ C^4 \end{array}$$

The resulting cost-centre-stack table will, perhaps surprisingly, be as follows:

$$\begin{array}{ccc} MAIN^1 \leftrightarrows A^2 \leftrightarrows B^3 \leftrightarrows C^5 \\ \Downarrow\uparrow & & \Downarrow \\ C^4 & \leftrightarrows & B^6 \end{array}$$

The resulting cost-centre-stack code is 6. The subtlety lies in the fact that from the cost-centre-stack code 5 a new cost-centre stack is built, 6, with B as the top cost centre. The duplicate copies of the cost centre B are avoided in this cost-centre stack by the fact that the back pointer from cost-centre stack 6 points to 4 and not 5. This results in the stack $\langle B, C, A, MAIN \rangle$ and not $\langle B, C, B, A, MAIN \rangle$, results which were expected. This maintains the compressed stacks.

Second subcase:

A second subcase is introduced with the following example:

Push the cost centre B onto the cost-centre stack 4 in the following cost-centre-stack table:

$$MAIN^1 \leftrightarrows A^2 \leftrightarrows B^3 \leftrightarrows C^4$$

The cost centre B already exists on the cost-centre stack. According to our rules introduced by the compressed stack scheme, only one instance of the cost centre can appear in the cost-centre stack at any one time.

The previous stack pointers make such an operation possible. They allow pre-existing stacks to be located and new stacks to be created if necessary. It is now feasible to follow the previous stack pointers back and check to see if the cost centre B had already been pushed on the stack. (This checking process will also have taken place in case 2, although an assumption that 'the cost centre has not been pushed on that stack before' was possible there, since the previous stack pointers are followed back until the first cost centre on the stack is reached).

Once the previous reference to B has been identified, the new stack is built with the previous occurrence of the cost centre effectively removed from the stack. This results in the cost-centre stack 6 in the following cost-centre-stack table:

$$\begin{array}{ccccc} MAIN^1 & \leftrightarrows & A^2 & \leftrightarrows & B^3 \leftrightarrows C^4 \\ & & \Downarrow\uparrow & & \Downarrow \\ & & C^5 & \leftrightarrows & B^6 \end{array}$$

Again from cost-centre stack 4 the cost centre B is added without a back pointer to the cost-centre stack 4. Instead it points to a cost-centre stack 5, an intermediate stack which needed to be built to support the new cost-centre-stack table. The cost-centre stack 6 produced in this case is $\langle B, C, A, MAIN \rangle$, the intermediate stack which is built is $\langle C, A, MAIN \rangle$.

It is possible that this intermediate stack will have needed to be built at some previous stage in the program's execution. If so, then the stack will already exist, as in subcase 1 above. Of course it might be possible that such a stack will not need to be built. Intermediate stacks can therefore be constructed with reduced storage, without explicit profiling details such as time and heap usage.

Third subcase

It is possible that once the occurrence of the new cost centre in the cost-centre

stack has been identified, the resulting new stack already exists[2]. A reference to this stack is produced as the result.

Extending the example seen above,

$$\begin{array}{ccccc} MAIN^1 & \leftrightarrows & A^2 & \leftrightarrows & B^3 \leftrightarrows C^4 \\ & & \Downarrow\uparrow & & \Downarrow \\ & & C^5 & \leftrightarrows & B^6 \end{array}$$

and pushing the cost centre C onto the cost-centre stack 6 produces the cost-centre-stack code 4 in the following cost-centre-stack table.

$$\begin{array}{ccccc} MAIN^1 & \leftrightarrows & A^2 & \leftrightarrows & B^3 \leftrightarrows C^4 \\ & & \Downarrow\uparrow & & \Downarrow \\ & & C^5 & \leftrightarrows & B^6 \end{array}$$

From the cost-centre stack $\langle B, C, A, MAIN\rangle$ the function C is called again. The crude approach to creating the new cost centre would have been to remove C from the stack and build the stack $\langle C, B, A, MAIN\rangle$ again, however, this is not necessary as this stack already exists. The previous stack pointers allow the location of the stack $\langle A, MAIN\rangle$, $\langle B, A, MAIN\rangle$, and finally the cost-centre stack $\langle C, B, A, MAIN\rangle$ to be identified, all of which exist in this case.

The previous stack pointers allow 'the stack which came before' to be identified. They also allow previous occurrences of cost centres to be found, and found in an economical manner. The backtracking procedure which has been described need only be followed until the previous occurrence of a cost centre is found; the cost centre may be discovered through the first previous stack pointer (in a recursive call to a function), or after fifty previous stack pointers, although it is certain that if the previous stack pointers are exhausted and the first cost centre is found then the cost centre has not been pushed before. Most of the time the backtracking will not need to go all the way back to the first cost centre, as the previous stack pointers ensure that the search is done as efficiently as possible.

If n is the number of cost centres in the current cost-centre stack and m is the

[2]This is a slightly more complicated example of case 1.

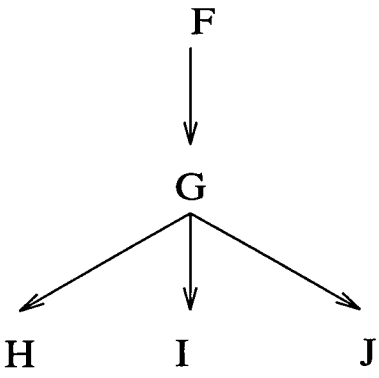


Figure 6.2: Example call-graph.

total number of entries in the cost-centre index tables for all n_i, then the complexity of this last case is $O(n + m)$.

6.3.2 Two examples

Consider the call-graph in Figure 6.2. The current cost-centre stack, after calls from F to G and then successful calls from G to H and I, would simply be $\langle G, F \rangle$ at some instance in time when I had been executed and G was still executing. The corresponding index table to this cost-centre stack would have references to the functions H and I and pointers to their corresponding cost-stack codes.

The cost-stack code 2 may represent this cost-centre stack in the cost-centre-stack table:

$$\begin{array}{ccccc} F^1 & \leftrightarrows & G^2 & \leftrightarrows & H^3 \\ & & \Downarrow\uparrow & & \\ & & I^4 & & \end{array}$$

The cost-stack table will also include references to the cost-stack codes 3 and 4.

> Cost-centre-stack code 3 represents the cost-centre stack $\langle H, G, F \rangle$
>
> Cost-centre-stack code 4 represents the cost-centre stack $\langle I, G, F \rangle$

These codes, plus the cost-stack code 1 for the cost-centre stack $\langle F \rangle$ make up the cost-stack table which we will refer to as T.

On calling function J, the operation $Push(J, 2)$, the cost-centre-stack system will:

1. Look up the cost-stack code 2 in the cost-stack table and return the state.
2. Determine whether its index table has a reference to the cost centre.
3. If there is a reference to the cost centre, then the pointer to the corresponding cost-stack code is returned as the new cost-centre stack. Function ends.
4. Otherwise, the previous stack pointers are followed back to determine whether there is a previous reference to the cost centre in the cost-centre stack. The code of each cost-centre stack visited is recorded on a conventional stack until the previous reference to the cost centre is identified or the top cost-centre stack, $\langle MAIN \rangle$, is reached. If the pushed cost centre is identified, its parent state is found; from here the recorded list is followed again in ascending order, creating new cost-centre stacks where necessary. At best this cost-centre stack already exists, in which case its cost-centre-stack code is returned as the current cost-centre stack; function ends. At worst we must create a new cost-centre stack, stage 6, the new code of which is returned as the current cost-centre-stack code.
5. If there are no previous references to the cost centre in the cost-centre stack then a new code is allocated and the cost-centre stack is copied to the new item in the cost-stack table, pushing the new cost centre onto the front of the stack. An index table is created in this new instance containing a back pointer to the parent cost stack, and this new cost-stack code is returned as the new current cost stack. Function ends.
6. Stage 5 is repeated for all cost centres on the conventional stack. When this is empty the cost-centre stack is returned as the result. Function ends.[3]

[3]The code for these steps is shown in Appendix B.

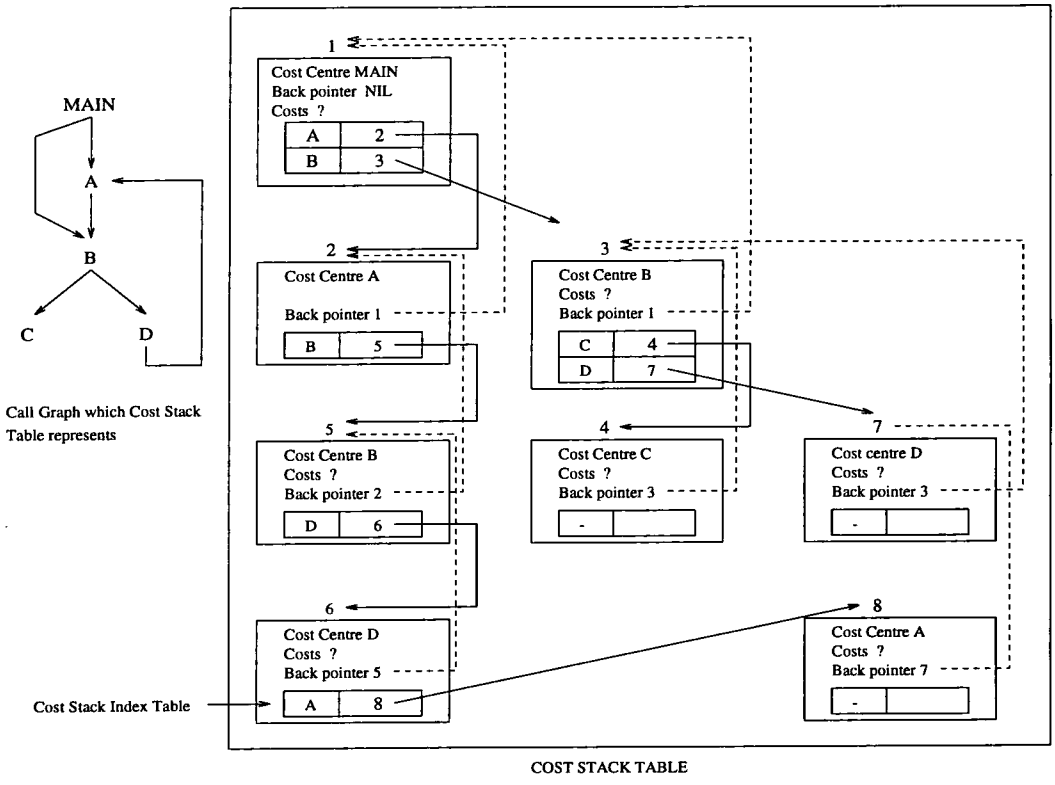


Figure 6.3: Example cost-centre-stack table.

The resulting situation in the example is:

$$\begin{array}{ccccc} & & J^4 & & \\ & & \Uparrow\downarrow & & \\ F^1 & \leftrightarrows & G^2 & \leftrightarrows & H^3 \\ & & \Downarrow\uparrow & & \\ & & I^4 & & \end{array}$$

The example considers function calls in a simple situation. It is more likely that a call-graph of a program execution is far more complicated, with calls to previously called functions and recursive definitions. The cost-stack implementation responds equally well in such situations. Consider the call-graph and cost-stack table in Figure 6.3.

An interesting case is presented when function D calls function A which is already in the current cost stack, state 6 in the example cost-stack table. The cost-stack rules state that a function can only appear in the cost stack once ($\mathcal{A}5$). It is therefore necessary to delete the previous reference to function A and create a new cost stack.

The back pointers are followed until cost centre A is found, creating as this process is taking place a list of the descending cost centres.

From the current cost stack indexed by the number 6 in Figure 6.3, the back pointers are followed starting with the current function at the head of the stack D. The descending stack $\langle B, D\rangle$ is recorded before the previous call to A is found. This previous reference to A is not added to the descending stack and its parent function $MAIN$ is found. From the $MAIN$ function the descending stack $\langle B, D\rangle$ is unwound to create the new cost stack. It is quite possible that part, or all, of the new cost stack already exists. For instance in the example the function B has already been called from the function $MAIN$ and this cost stack already exists. Likewise the function D has already been called from the function B so there is no need to create a new cost stack $\langle D, B, MAIN\rangle$. Finally the function call to A is added to the cost stack. This creates state 8 in the figure.

There will in general be many cases when the cost-centre stack which is referenced already exists and it is only the exception when a new stack must be created. For most of the time, therefore, updating the current cost-centre stack simply means using a reference to another cost-centre stack in the cost-centre-stack index table; the expense is simply the look-up in the cost-centre table, an operation which is easily optimised.

6.4 Integration with GHC

The cost-centre-stack profiler is implemented on the Glasgow Haskell Compiler version 0.22. The cost-stack code is written in C and is included with the GHC run-time code. A large amount of effort has been spent on making this code efficient.

The implementation is based upon a number of components.

- Though it need not always be the case, it is sensible that the source-code expressions are identified using the `-auto-all` compiler profiling option. This

ensures that all top-level functions are identified as cost centres and included in the profiling results. Should the programmer annotate the code with cost centres by hand, the information supplied to the post-processor will be less detailed[4]. The results themselves may be equally valid but fewer cost centres will be included during post-processing.

- The *current cost-centre stack* is maintained throughout the program execution.
- The run-time system is extended to include the cost-centre-stack table.
- Compiler optimisations are maintained within the cost-centre-stack context.
- A post-processor manipulates the profiling results according to the programmer's requirements.

These components are discussed in turn.

6.4.1 Identifying source-level expressions

During profiling cost centres are set using the `-auto-all` run-time flag. This records profiling costs in terms of all top-level functions. The advantage of such a scheme is that all top-level functions are made available for post-processing. The programmer is able to follow the cost-centre stack function by function until the expensive part of the code is identified.

Adding further cost centres will simply add more detail to the cost-centre stacks. Adding cost centres to local definitions as well as all top-level functions will increase the granularity of the stacks, that is the stacks will include references to the cost centres describing local and global definitions. Reducing the number of cost centres when profiling will simply reduce this effect.

[4]Unless the programmer places more cost centres in the code that the `-auto-all` profiling flag.

6.4.2 Maintaining the current cost-centre stack

Costs must be attributed to the correct cost-centre stack during execution. This is achieved by recording in a current cost-centre-stack register the cost-centre stack which, in the context of the expression, is currently being executed. This register is also used to return the resulting cost-centre stack cc_z with the result of an expression. Any costs incurred within the expression are attributed to the current cost-centre stack.

6.4.3 The extended run-time system

Many of the abstract machine-level modifications are implemented by a selection of simple, low-level, code modifications to the existing cost-centre profiler of the Glasgow Haskell Compiler.

Cost-centre-stack program code

The interface of the cost-stack code with the remainder of the compiler is supported by two functions, the *push* operation, for pushing new cost centres onto the cost-centre stack and the *print* operation which allows cost-centre stacks and their associated costs to be printed.

A cost-centre stack is implemented as a pointer to the cost-centre label at the head of the stack[5], a pointer to the cost-stack index table, a pointer to the previous stack, and integers recording the time and heap allocated to that cost centre.

A cost-centre stack therefore contains 3 pointers and two integer values for time and heap usage. This is designed in such a way as to minimise the profiling overheads. This is equivalent to approximately 20 bytes per entry[6] on the cost stack. Cost-centre-stack codes are effectively replaced by pointers to improve the

[5]This label is part of the already existing cost-centre information. It is possible to access the location in memory where this label is stored. The overheads involved in storing this label for the cost-centre stacks are therefore only the price of a pointer.

[6]Based on gcc version 2.5.8.

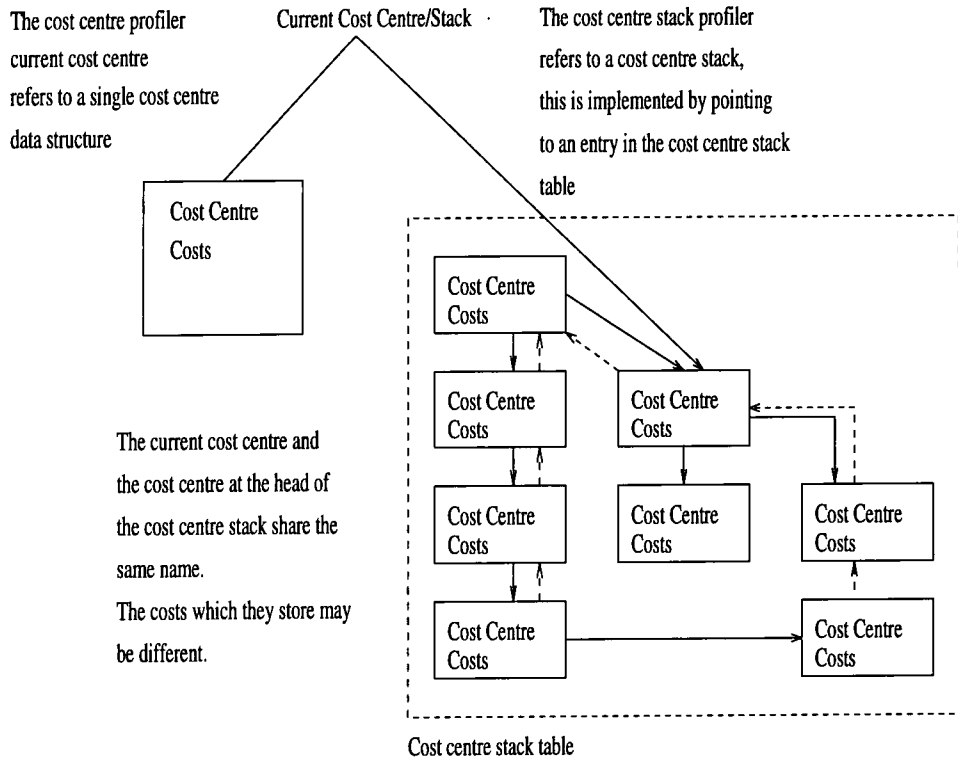


Figure 6.4: Implementation of the current cost-centre stack.

efficiency of the implementation.

The cost-stack index table contains a pointer to a cost-centre label, a pointer to the associated cost-centre stack and a pointer to the next cost-centre index table item. This structure therefore contains 3 pointers (12 bytes) for each item stored in it.

In the results chapter the effect of this implementation is discussed, particularly whether the considerations made when designing cost-centre stacks actually produce an effective implementation.

Altering the GHC cost centres

The discussion of semantic definitions in section 6.2.1 shows the movement from the current cost centre to the current cost-centre stack. This change was implemented at a low level by changes to the current cost-centre code supported by C macros in the GHC compiler.

In Figure 6.4 the modifications made to the run-time system can be seen. The implementation of the current cost centre in the cost-centre compiler pointed to a single cost centre; this can be seen to the left of the figure. To the right of the figure are the extensions made. The current cost centre (now appropriately renamed 'current cost-centre stack') points to an entry in the cost-stack table; this entry is the current stack.

It would be possible to remove the old cost centre from the current cost-centre stack, thus directly replacing the notion of a current cost centre with a current cost-centre stack. However, the current cost centre is retained. This leaves open the possibility of improving the efficiency of the cost-centre-stack profiler, but for the time being additions are made to the existing profiler rather than any of the previous information being removed.

It may have been noticed that there is no reference to the cost-centre-stack code in the implementation. In fact this code is replaced by an actual code in memory where the cost-centre stack is stored. In other words, it is represented as a pointer! Now it does not matter how many stacks are created, the cost-centre-stack code will always be 4 bytes.

Each cost centre is registered at run-time by traversing the module dependency graph at the start of execution. In each module a small routine is declared which registers the cost centres declared in that module; it also calls the registering routine of all the imported modules, thus ensuring the registration of all modules in the system. The declaration of a cost centre calls a small piece of code, `CC_DECLARE`, to reset the cost counts. This registering procedure is used in the cost-centre profiler so that all cost centres in the code are declared.

The current cost-centre stack is initialised to $\langle\rangle$. The cost-centre-stack table is built dynamically as the execution of the program proceeds. The cost-centre stacks are implemented in a single module written in C. This resets the appropriate counters and labels as the cost-centre stacks are built.

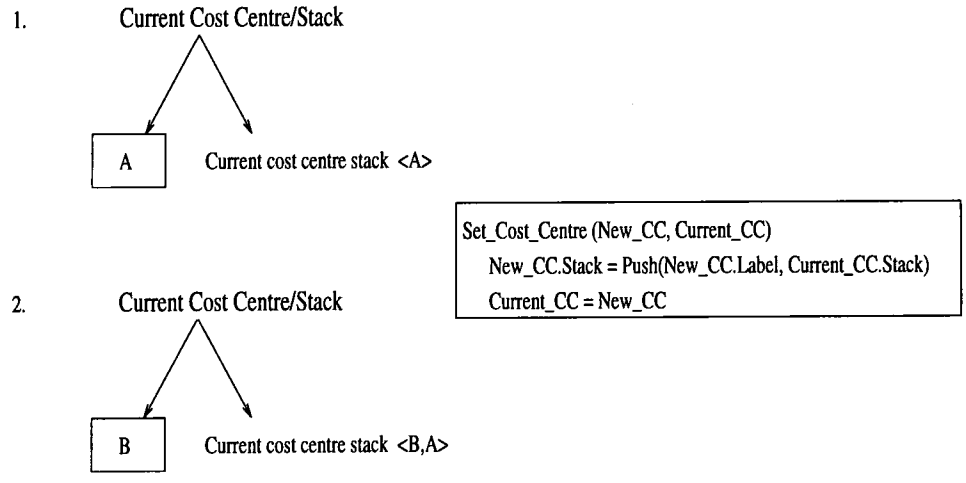


Figure 6.5: Implementation of Push.

The implementation of the Push operation

The cost-centre-stack profiling scheme is implemented by adding the notion of a cost-centre stack to the cost-centre profiler in the STG-machine. All costs of function evaluation are attributed to the current cost-centre stack.

In the implementation of the cost-centre profiler an execution of an `scc` expression would set a new current cost centre. The old cost centre would be saved on the return stack so that when a constructor was restored, the correct cost centre would be returned as the current cost centre. In this way the scoping of the cost centres was achieved.

Profiling using cost-centre stacks rather than just cost centres works in a similar way. The addition to the cost-centre system is that when a cost centre is set it is also pushed onto the current cost-centre stack.

Figure 6.5 shows how this works. At stage 1 in the diagram the current cost centre/stack refers to the cost centre A; the current cost-centre stack is also referred to, in this case it is $\langle a \rangle$.

When the current cost centre is set (the pseudo code 'Set_Cost_Centre' in the diagram) the current cost centre and the current cost-centre stack are updated. Updating the current cost-centre stack uses the push operation.

The new cost-centre stack is created with the operation $push(B, \langle A \rangle)$, resulting in the cost-centre stack $\langle B, A \rangle$.

Recording costs

The underlying mechanism for recording costs in the GHC profiler is implemented using UNIX signals. The execution of the program is interrupted every 20ms using the `setitimer` system call. Each of these interrupts is handled using the `signal` system call, during which the interrupt handler increments the `time_tick` counter of the current cost centre and also of the current cost-centre stack.

When the cost centres and their results are printed at the end of a program's execution, the cost stacks are also printed. A discussion of the results is left until later in the chapter.

6.5 Maintaining Compiler Optimisations

Some of the first results from the cost-centre-stack profiler were recorded without any further changes to the Haskell compiler. The results showed the interesting behaviour of the Glasgow compiler optimisations which were performed on the code.

Consider the call-graph in Figure 6.6, which is derived from one of the test programs. It is quite clear to the user which call stack refers to a certain part of the program. For instance, a visit to the function g would have come from stacks $\langle d, b, a, main \rangle$ or $\langle e, b, a, main \rangle$. The user would not expect anything different.

The first results however included the stack $\langle e, a \rangle$. From looking at the call-graph it is clear that such a stack should not exist, unless of course it in fact represented the stack $\langle e, b, a, main \rangle$, which to all accounts it did. The complete set of initial results are shown for reference[7]:

[7]These results will be discussed in more detail in chapter 7.

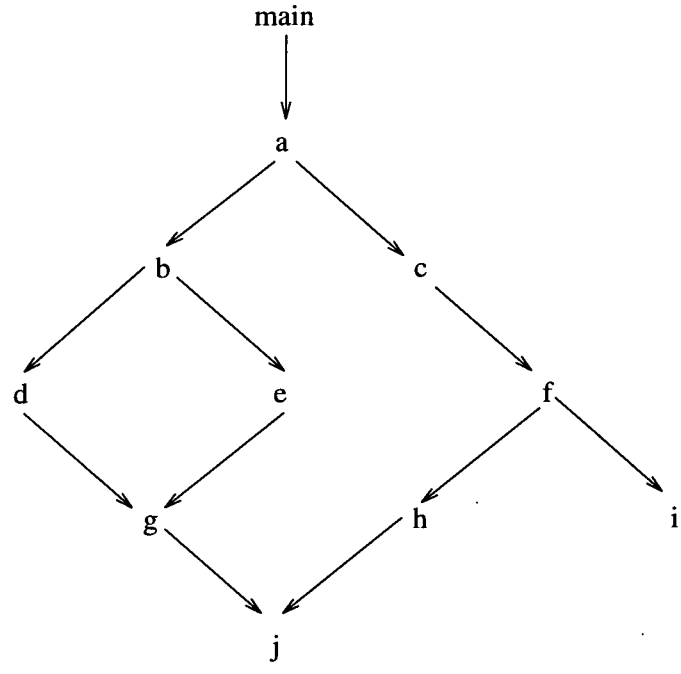


Figure 6.6: Call-graph of an experimental program.

```
A cost stack <Main_j,Main_h,Main_f,Main_c,> with      1462 TICKs   X
A cost stack <Main_f,Main_c,> with                     353 TICKs   X
A cost stack <Main_i,Main_f,Main_c,> with                8 TICKs   X
A cost stack <Main_h,Main_f,Main_c,> with                0 TICKs   X
A cost stack <Main_g,Main_e,Main_a,> with               12 TICKs   X
A cost stack <Main_j,Main_g,Main_e,Main_a,> with        12 TICKs   X
A cost stack <Main_a,> with                             20 TICKs
A cost stack <Main_e,Main_a,> with                       0 TICKs   X
A cost stack <Main_b,Main_a,> with                      14 TICKs
A cost stack <Main_d,Main_b,Main_a,> with                0 TICKs
A cost stack <Main_g,Main_d,Main_b,Main_a,> with        25 TICKs
A cost stack <Main_j,Main_g,Main_d,Main_b,Main_a,> with 28 TICKs
A cost stack <MAIN,> with                                0 TICKs
A cost stack <Main_c,> with                              0 TICKs   X
```

The results contain a collection of 'broken stacks', annotated in the results with the extra column and the Xs. These broken stacks contain instances of the form $\langle z, x \rangle$ or $\langle z, y \rangle$ where they should have been represented as $\langle z, y, x \rangle$.

The program was compiled and executed with the $-O$ flag and consequently the cost stacks were being incorrectly constructed. This could be tested by switching

the optimiser off. The non-optimised program produced the following results[8]:

```
A cost stack <Main_j,Main_h,Main_f,Main_c,Main_a,> with  1429 TICKs
A cost stack <Main_f,Main_c,Main_a,> with                 359 TICKs
A cost stack <Main_j,Main_g,Main_e,Main_b,Main_a,> with    13 TICKs
A cost stack <Main_g,Main_e,Main_b,Main_a,> with           10 TICKs
A cost stack <Main_a,> with                                 0 TICKs
A cost stack <Main_i,Main_f,Main_c,Main_a,> with            9 TICKs
A cost stack <Main_b,Main_a,> with                          0 TICKs
A cost stack <MAIN,> with                                   0 TICKs
A cost stack <Main_h,Main_f,Main_c,Main_a,> with            0 TICKs
A cost stack <Main_g,Main_d,Main_b,Main_a,> with           23 TICKs
A cost stack <Main_j,Main_g,Main_d,Main_b,Main_a,> with    32 TICKs
A cost stack <Main_d,Main_b,Main_a,> with                   0 TICKs
A cost stack <Main_c,Main_a,> with                          0 TICKs
A cost stack <Main_e,Main_b,Main_a,> with                   0 TICKs
```

These demonstrate that without the compiler optimisations the cost-centre stacks produced are un-broken and therefore correct. The implementation of an accurate post-processor relies on the fact that cost-centre stacks are complete; any post-processing would be very difficult if the cost-centre stacks needed to be patched or repaired. The results produced may also be inaccurate or misleading.

The development of the cost-centre-stack profiler could have relied on the fact that the code could only be compiled or executed without compiler optimisations. The idea of profiling a system without any optimisations does however seem rather contradictory, as, assuming that the programmer would wish to run his code with the optimisations present, the code executed would be different to the code profiled.

It was therefore necessary to consider changes to the compiler optimiser to maintain correctly-constructed cost-centre stacks. These changes aimed as much as possible at retaining the existing optimisations to the code and producing an executable which was comparable in size and execution speed to the original cost-centre profiled optimised code.

[8]The function `main` was not profiled.

6.5.1 Lazy evaluation

Lazy evaluation forces the interleaving of the evaluation of an expression with the evaluation of the inputs which it demands. Since an expression itself is in demand, it will also be interleaved with the execution of its demander. For this reason, stated in Sansoms's thesis, the resulting order of execution does not have a direct correspondence with the code to which the results must be mapped. The call stacks will not therefore be explicit call stacks at run-time but rather 'demand' stacks.

These demand stacks will accurately record the order of evaluation in the program. It should be considered, however, that optimisations performed by the compiler may affect the order of the evaluation. These fall into two categories: firstly, hidden functions are introduced by a high-level translation of syntactic sugars such as list comprehensions; and secondly, auxiliary functions and definitions are introduced as expressions are transformed.

The combined effect of all the transformations may change the structure of the original source code. These code transformations may therefore change the appearance of the cost-centre stacks. The creation of the stack $\langle e, a \rangle$, found in the earlier results, is in fact an optimised demand stack.

The compiler transformations, presented in chapter 5, are re-examined and redefined to work correctly in the context of the cost-centre stacks. The correctness of the new transformations in the preservation of cost scoping is also investigated.

6.6 Compilation by transformation revisited

Recall the optimisation rules investigated in chapter 5. The first transformation rule explored, $T1$, floated an `scc` expression into a `let`-binding.

$(T1)$ `scc` cc `let` $v = E_v$ in $E \quad \Rightarrow$ `let` $v = (\texttt{scc}_{sub}\ cc\ E_v)$ in (`scc` cc E)

The cost of reducing the expression E remains within the scope of the cost centre `scc`. The costs of the expression E_v must also be kept within the scope of the cost

centre *cc*. This is done using the $\texttt{scc}_{sub}$ annotation to indicate that although this sub-expression has been moved into the scope of a different cost centre, the costs of evaluation must still be attributed to the cost centre *cc*. Evaluating an $\texttt{scc}_{sub}$ expression does not increment the count of the expression instances evaluated, this is only incremented when the original `scc` expression is entered. Therefore both the costs of evaluating E and E_v are attributed to the cost centre *cc*.

($T2$) $\texttt{let}\ v\ =\ E_v\ \text{in}\ (\texttt{scc}\ cc\ E)\ \Rightarrow\ \texttt{scc}\ cc\ \texttt{let}\ v\ =\ (\texttt{scc}_{sub}\ ecc\ E_v)\ \text{in}\ E$

The second transformation, $T2$, floated a `let`-binding into an `scc` annotation. The `let`-binding and the expression E are in the scope of the cost centre *cc* on the right of the transformation rule. The right hand side of the transformation must also be annotated with a $\texttt{scc}_{sub}$ expression, enclosing the expression E_v with the enclosing cost centre (*ecc*). This prevents the costs of evaluating the expression E_v from being incorrectly attributed to the cost centre *cc*.

It was noted that this second transformation can only be performed if the enclosing cost centre *ecc* is known.

The transformation, $T3$, was applied by the simplifier to remove the redundant cost centres in the code.

($T3$) If E_v has an `scc` expression on it then $\texttt{scc}_{sub}\ cc\ (\texttt{scc}\ cc'\ E_v)\ \Rightarrow\ \texttt{scc}\ cc'\ E_v$

This transformation preserves the immediately enclosing cost centre, but removes any of the other cost centres, as in the cost-centre profiler these are seen as redundant.

It is this transformation which is in fact responsible for the compiler optimiser breaking the cost-centre stacks of the cost-centre-stack profiler.

6.6.1 The source of broken stacks

The transformation, $T3$, preserves the immediately enclosing cost centre within optimised code but, as the results of the cost-centre-stack profiler show, prevents the stack of cost centres from being accurately recorded. This optimisation creates

the broken stacks in the initial results.

It is therefore possible to accurately record the cost-centre stacks of a program if the optimisation transformation, $T3$, is removed. The `let`-floating code in the simple Core language of GHC is altered to reflect this analysis.

As expected the results received from the modifications to the compiler optimiser show the correct construction of cost-centre stacks. The compiler optimiser remains active throughout the compilation and execution of this program. The results are:

```
A cost stack <Main_j,Main_h,Main_f,Main_c,Main_a,> with  1471 TICKs
A cost stack <Main_f,Main_c,Main_a,> with                 366 TICKs
A cost stack <Main_j,Main_g,Main_e,Main_b,Main_a,> with    15 TICKs
A cost stack <Main_g,Main_e,Main_b,Main_a,> with           11 TICKs
A cost stack <Main_a,> with                                 0 TICKs
A cost stack <Main_b,Main_a,> with                          0 TICKs
A cost stack <Main_i,Main_f,Main_c,Main_a,> with           10 TICKs
A cost stack <MAIN,> with                                   1 TICKs
A cost stack <Main_c,Main_a,> with                          0 TICKs
A cost stack <Main_g,Main_d,Main_b,Main_a,> with           26 TICKs
A cost stack <Main_j,Main_g,Main_d,Main_b,Main_a,> with    34 TICKs
A cost stack <Main_d,Main_b,Main_a,> with                   0 TICKs
A cost stack <Main_e,Main_b,Main_a,> with                   0 TICKs
A cost stack <Main_h,Main_f,Main_c,Main_a,> with            0 TICKs
```

The transformation, $T3$, is not an essential transformation operation. It was included in the original GHC code to remove any unnecessary `scc` annotations in optimised code. Removing this transformation will therefore have a minimal effect on the efficiency of the resulting code.

It is necessary to consider whether removing this transformation will have any effect on the scoping of cost centres, that is whether the program costs will be correctly attributed to the cost-centre stacks according to the cost-centre-stack semantics specified earlier.

6.6.2 Preserving the semantics of cost-centre stacks

The compiler optimisations are modified so that they preserve the embedding of the cost centres, thus enabling the correct construction of cost-centre stacks. As well as identifying the correctness of the embedding of cost centres it is also necessary to consider the cost-scoping behaviour of the cost-centre stacks.

Altering the compiler optimisations may preserve the order in which the cost centres are pushed onto the cost-centre stacks, but for the costs to remain accurate the costs must also be assigned to the cost-centre stacks correctly.

The investigation follows the effect of the let-floating transformation rules described earlier on a stack of cost centres. These compiler transformation rules have been analysed in terms of single enclosing cost centres for the cost-centre profiler; this section endeavours to judge whether the rules preserve the scope of the cost centres in the context of cost stacks.

The correctness of the transformation rules applied to a cost-centre stack of size 1 is unquestionable, this is just the correctness of the transformation rules as applied to single cost centres. This work has been considered in [Sansom, 1994]. The more interesting case is the analysis applied to a cost-centre stack greater than size one. The exploration is extended and applied in the context of two enclosing cost centres, thus presenting the compiler transformations in terms of a cost-centre stack of size 2 or more[9].

In each left-hand side of the transformation rules there are three important constituents; the cost centre, the value inside the let-binding expression E_v and the expression E itself. The investigation of correctness examines the effect of applying the transformations $T1$ and $T2$ to the expressions with the surrounding cost-centre stack $\beta^\frown\alpha^\frown S$. α and β represent any cost centres and S represents any cost-centre stack. In each case the cost-centre stacks surrounding the expressions E_v and E, before and after the transformations have been applied, should be equivalent.

[9]This leaves open the possibility of a complete mathematical proof on the correctness of the transformations over a stack of size n. The base case is supplied by Sansom, the inductive case is supplied in this thesis.

Analysis of `let`-floating transformation ($T1$)

From the `scc` expressions it can be seen that the cost-centre stack over E_v before the transformations is $\beta^\frown\alpha^\frown S$, over E it is also $\beta^\frown\alpha^\frown S$:

$$\texttt{scc}\ \alpha\ (\texttt{scc}\ \beta\ (\texttt{let}\ v\ = E_v\ \text{in}\ E))$$

Applying $T1$ over β

$$\texttt{scc}\ \alpha\ (\texttt{let}\ v\ = (\texttt{scc}_{sub}\ \beta\ E_v)\ \text{in}\ (\texttt{scc}\ \beta\ E))$$

Applying $T1$ over α

$$\texttt{let}\ v\ = (\texttt{scc}_{sub}\ \alpha\ (\texttt{scc}_{sub}\ \beta\ E_v))\ \text{in}\ (\texttt{scc}\ \alpha\ (\texttt{scc}\ \beta\ E))$$

The cost-centre stack over E_v after the transformations is $\beta^\frown\alpha^\frown S$, over E it is $\beta^\frown\alpha^\frown S$.

This demonstrates that the transformation $T1$ preserves the semantics for the cost-centre stacks when the transformation $T3$ is removed.

Preserving the cost-centre stack requires $T3$ to be removed, otherwise the result is the enclosing cost centre β, and not the cost-centre stack $\beta^\frown\alpha^\frown S$. With $T3$ removed the correctness of the cost-centre-stack semantics is preserved [10].

□

Analysis of `let`-floating transformation ($T2$)

The same analysis is performed on the transformation $T2$. The cost-centre stack is also represented as $\beta^\frown\alpha^\frown S$.

Cost-centre stack over E_v before the transformations $= S$, over $E = \beta^\frown\alpha^\frown S$

$$\texttt{let}\ v\ = E_v\ \text{in}\ (\texttt{scc}\ \alpha(\texttt{scc}\ \beta\ E))$$

Applying $T2$ over α

[10] It is recognised that the reduction of the `let` is moved outside the scope of the cost centres. This is the same problem which was recognised in the cost-centre profiler and considered to have little effect on the results. We therefore make the same assumption that the movement of this single reduction between cost-centre stacks is not considered to affect the results in any significant way.

$$\texttt{scc}\ \alpha\ (\texttt{let}\ v\ =\ (\texttt{scc}_{sub}\ S\ E_v)\ \text{in}\ (\texttt{scc}\ \ \beta\ E))$$

Applying $T2$ over β

$$\texttt{scc}\ \alpha\ (\texttt{scc}\ \beta\ (\texttt{let}\ v\ =(\texttt{scc}_{sub}\ \alpha\ (\texttt{scc}_{sub}\ S\ E_v))\ \text{in}\ E))$$

Cost-centre stack over E_v after the transformations $= \alpha^\frown S$, over $E = \langle\beta, \alpha\rangle^\frown S$. Therefore in this case the transformation $T2$ does not preserve the enclosing cost-centre stacks. It is not correct to have α in the cost-centre stack of the expression E_v. When the costs of evaluating E_v are inherited to the cost centres in the cost-centre stack in the scope of E_v they will incorrectly be attributed to α. The cost centre α did not begin in the scope of the expression E_v and the transformation $T2$ moves it into the scope of this cost centre, leading to potentially misleading results.

□

The transformation $T2$ will not correctly preserve the cost-centre-scoping semantics when used in the context of the cost-centre-stack profiler on optimised code. To find some way of solving this problem it is necessary to understand in what context this second rule is used as moving the `scc` outside of the let expression seems to be contradictory to rule $T1$.

6.6.3 Cost-centre-stack transformation rules

The transformation rules $T1$ and $T2$ allow lets to be floated out and floated in respectively, both of these are desirable depending upon the situation.

When performing let-floating operations there are two basic ideas to be respected. Firstly, it is always good to float lets out past lambdas (loop reduction), and secondly, it is best to float them in as far as possible (provided that sharing is not lost) as allocation will be avoided if an alternative path is taken. It is desirable therefore to be able to propagate lets into and out of `scc`'s.

The cost-centre-stack work is supported by the fact that at present the Glasgow Haskell Compiler does not perform the let-floating transformation $T2$ as part of its

compiler optimisations; instead the propagation of lets into an scc is prohibited.

It may seem strange that the transformation $T2$, despite being documented, is not implemented as part of GHC. Propagating lets into scc's is still a desirable activity, but there is no way in the Glasgow compiler of keeping track of the enclosing cost centre *ecc*. Therefore, although this optimisation was able to be suggested, no method of implementation was achieved.

However, with the cost-centre-stack theory this problem can be solved. With cost-centre stacks it is not necessary to keep track of the enclosing cost centre *ecc* since it will already be on the stack. A modified version of $T2$, within the context of cost-centre stacks, can be defined.

$$(T2\ \textit{stacks})\quad \texttt{let}\ v\ =\ E_v\ \text{in}\ (\texttt{scc}\ cc\ E)\ \Rightarrow\ \texttt{scc}\ cc\ \texttt{let}\ v\ =\ (\texttt{scc}_{pop}\ E_v)\ \text{in}\ E$$

Here '$\texttt{scc}_{pop}\ E_v$' means evaluate E_v in the context of a cost-centre stack with the current cost centre, *cc*, popped off the top of the stack. As the cost-centre stack is recorded at run-time the compiler problem is solved, as it is not necessary to know the *ecc* at compile time. This new rule also preserves the cost-stack-scoping rules which $T2$ did not.

As well as extending the methods of profiling available, the cost-centre-stack profiling theory may also have other benefits; in this case allowing proposed optimisation transformations to be implemented in practice.

These transformations supported by the notation of cost-centre stacks can be extended. A valid optimising transformation which would arise if a let was floated in and then out again would be:

$$(T4\ \textit{stack})\quad \texttt{scc}_{sub}\ cc\ \ (\texttt{scc}_{pop}\ e)\ \Rightarrow\ e$$

Another proposed transformation could be:

$$(T5\ \textit{stack})\quad \texttt{scc}_{out}\ cc\ \ (\texttt{scc}_{pop}\ e)\ \Rightarrow\ \texttt{scc}_{pop2}\ e$$

This, for example, would evaluate e with two cost centres popped off the stack. There are a number of possibilities with this new approach.

6.6.4 The cycle problem

Although the above scheme does seem promising this is not the end of the story. Using compressed stacks means that it is not true that $\forall s \bullet \forall e \bullet pop(push(e, s)) = s$; that is, a push followed by a pop does not always get you back to the original stack.

Consider this example. Beginning with the cost-centre stack $\langle b, a \rangle$ the operation $push(a, \langle b, a \rangle)$ produces the cost-centre stack $\langle a, b \rangle$. Applying the operation $pop(\langle a, b \rangle)$ to this cost-centre stack returns $\langle b \rangle$. Conventionally we would expect this to return the original cost-centre stack, but because of stack compression this is not the case.

The rule (*T2 stacks*) will not work if the $\texttt{scc}_{pop}$ operation is defined as a standard stack pop, unless the `scc` on the left hand side of the rule is forced to create an uncompressed stack at the point at which *cc* is added.

If this is the case, for notational convenience, the `scc` on the right hand side of the (*T2 stacks*) rule is underlined to demonstrate that this represents an uncompressed rather than a compressed push.

(*T2 stacks*) $\texttt{let } v = E_v \text{ in } (\texttt{scc } cc\ E) \Rightarrow \underline{\texttt{scc}}\ cc\ \texttt{let } v = (\texttt{scc}_{pop}\ E_v) \text{ in } E$

This will preserve the uncompressed stack for the standard pop operation. Once the $\texttt{scc}_{pop}$ operation occurs then the nesting of the rules will ensure that the stacks then revert back to the system of compressed stacks.

It is therefore feasible to have a cost stack which is made up of compressed and non-compressed stacks.

This is a solution based on the fact that compressed stacks are not 'popable' because each cost centre only appears on the stack once.

There are some questions which this method presents:

- If the cost-centre-stack scheme is augmented with uncompressed stacks, will this mean that all the stacks are uncompressed? i.e. how often will this un-compression occur.

- Using uncompressed stacks requires the post-processing semantics to be re-defined to include multiple occurrences of cost centres. Uncompressed stacks will not then affect the correctness of the algorithm, though they will impose some extra overheads on the post-processing part of the code.

- Will the uncompressed stacks become huge? This may effectively create the problem which was first avoided by the creation of compressed stacks. Some compromise can be made to try and prevent the problem: For instance it is possible to avoid using the transformation rule (*T2 stacks*) when the let expression is recursive; having a locally recursive definition in the context of uncompressed cost stacks is potentially disastrous. It is possible to identify in the compiler whether the let is recursive or not and therefore restrict the algorithm to only those non-recursive cases.

The proposed solution to this problem is to optimise the compressed and uncompressed activity so that uncompressed stacks are only used within the `let` and not the `let-rec` expressions of optimised code.

The author is indebted to Patrick Sansom of Glasgow University for his advice regarding compiler optimisations.

6.7 Post-Processing Cost-Stack Results

The cost-centre stacks results are amenable to two forms of post-processing. Firstly, they can be used to produce an accurate inheritance profile and secondly, they can be used to select and deselect cost centres.

6.7.1 An accurate inheritance profile

A flat profile

Initially the cost-centre stacks are used to produce a flat time profile, similar to the results of the cost-centre profiler. This is displayed as a text file and also in

the graph-tool environment (see later section).

The flat profile is created according to the secondary cost semantics seen in section 6.2.2 using a script program written in C. As an example consider the following three stacks produced as output from the cost-centre-stack profiler:

```
<Main_f,Main_j,Main_MAIN,> with 10 Ticks
<Main_g,Main_j,Main_MAIN,> with 20 Ticks
<Main_f,Main_MAIN,>        with 15 Ticks
```

Calculating the flat profile involves collecting those cost centres at the head of each of the cost-centre stacks, `Main_f`, `Main_g` and `Main_f`, and adding up their associated costs. These results,

$$\texttt{Main\_f} = 25 \text{ Ticks}$$
$$\texttt{Main\_g} = 20 \text{ Ticks}$$

can then be calculated as a percentage of the total number of time ticks recorded; 55.55% and 44.45% respectively. Any remaining cost centres in the stacks are given zero costs.

The script program takes the file containing the cost-centre stacks as its input and after processing the costs it produces two text files as its output. The first is the flat profile of the following form:

```
Fri Nov  3 15:39 1995 Time and Allocation Profiling Report  (Final)
                 (Hybrid-Cost-Stack Scheme)

        run +RTS -pT -RTS

COST CENTRE      MODULE      GROUP          scc subcc  %time
Main_f           Main        Main             2     0   55.5
Main_g           Main        Main             1     0   44.4
Main_j           Main        Main             2     2    0.0
Main_MAIN        Main        Main             3     3    0.0
```

The second is the input to the graph-tool.

```
( object ) 0 0 0 0 0 ( Main_j )( cost ) ( 0.0 ) ( _ ) ( _ ) object
( object ) 1 0 0 0 0 ( Main_f )( cost )( 55.5 ) ( _ ) ( _ ) object
( object ) 2 0 0 0 0 ( Main_g )( cost )( 44.4 ) ( _ ) ( _ ) object
( object ) 3 0 0 0 0 ( Main_MAIN )( cost )( 0.0 )( _ )( _ ) object
( link ) 0 2 0 0 0 0 0 0 ( -1 ) ( directed ) ( LineSolid ) link
( link ) 0 1 0 0 0 0 0 0 ( -1 ) ( directed ) ( LineSolid ) link
( link ) 3 0 0 0 0 0 0 0 ( -1 ) ( directed ) ( LineSolid ) link
( link ) 3 1 0 0 0 0 0 0 ( -1 ) ( directed ) ( LineSolid ) link
```

This file produces enough information to allow a call-graph to be displayed. The `objects` in the file correspond to the nodes of the call-graph. These nodes are uniquely numbered so that links can be created between the nodes.

An inherited profile

The accurate inheritance profile is produced using the post-processing script file in accordance with the inheritance semantics of section 6.2.2. The previous three cost-centre stacks produce the inherited results:

`Main_MAIN` = 45 Ticks

`Main_j` = 30 Ticks

`Main_f` = 25 Ticks

`Main_g` = 20 Ticks

These can also be presented as percentage figures, 100%, 66.6%, 55.5%, 44.4% respectively. They can be displayed in a table, in a similar manner to the flat profile, and also in the graph-tool environment.

6.7.2 Selecting and deselecting cost centres

Cost centres can be selected and deselected within the post-processing tool. It is currently possible to toggle this facility so that the programmer can either select or deselect the cost centres according to whether he is interested in profiling them. This facility could easily be extended to include modules or groups of cost centres.

Those cost centres which are selected are included in the profiling results. In the above example, selecting the cost centres `Main_MAIN` and `Main_j` produces the

following inherited results:

`Main_MAIN` = 15 Ticks

`Main_j` = 30 Ticks

This accurately subsumes the cost of `Main_f` to the correct calling function. It also inherits all the costs below `Main_j` to that point in the program and no higher, unless `Main_MAIN` is directly responsible for its costs.

Running the post-processing script on even detailed cost-centre stacks only takes a matter of seconds to produce the required results.

6.7.3 Displaying call-graphs

A substantial amount of work has been done on the display of functional call-graphs by the Centre for Software Maintenance at the University of Durham, [Boldyreff, Burd and Hather, 1995] [Kinloch and Munro, 1994]. The AMES (Application Management Environments Support) project[11] has developed a suit of tools which allow the functionality of a program to be analysed interactively. This helps the programmer in the program comprehension process and also with application understanding, a broader concept which allows the programmer to understand not only the functioning of the code but also the functionality of the application itself.

The AMES programming tool includes a method of displaying call-graphs for programs [Bodhuin, 1995]. Both automatic and manual layout can be performed on these graphs; call-graphs can also be automatically manipulated within this tool. Operations allow call-graphs to be simplified and displayed hierarchically, progressive changes are stored and a history of the call-graphs during analysis is displayed within the NCSA Mosaic environment. The graph-tool is described in more detail in section 6.7.4.

This programming environment has been modified to include a method for

[11] ESPRIT Project: number 8156. Partners: Cap Gemini Innovation, Cap Programmator, Intecs Sistemi Spa, Matra Marconi Space, OPL-TT, Space Systems Finland, Valtion Teknillinen Tutkimuskeskus and University of Durham.

displaying the cost-centre-stack profiler results.

Cost-centre-stack profiling results

The call-graph results are displayed in graphical form. There are a number of points to note about this form of display.

- The graph is presented with function names at each individual node. These function names correspond to cost centres in the code, so as extra cost centres are added these will also appear as nodes in the call-graph. Function calls are represented by the arcs in the graph.
- The cost-centre name (function name) and the corresponding percentage of execution time are displayed at each node.
- The costs at each node of the graph may be inherited or non-inherited costs. These are selected by the programmer using the post-processing tool. Each time a post-processing function is run, the call-graph must be reloaded for the updated results.
- The most *expensive* cost-centre stack can be selected and highlighted. This is useful when the call-graph is very large. The programmer is immediately pointed towards that part of the program where the largest percentage of the costs manifest themselves. From this the programmer can work out not only where an expensive function in the call-graph is, but, if the function is shared, in which direction his attention should be focussed.
- The graph is interactive and may be arranged using the mouse. This enables parts of the graph that are interesting to the programmer to be brought into the foreground. The graph-tool also has a virtual display which allows parts of the graph to be displayed at any time.
- All functions on the graph-tool and post-processor are selected from the window based menu system. This gives the programmer control over how he manipulates and views the results (see next section).

- The post-processor, which controls the results to the graph-tool, is written in *Tk* and run under the *wish* interpreter.

6.7.4 Further facilities

A number of further facilities are provided to aid the programmer. Call-graphs can be simplified in a number of ways: specific nodes can be removed; specified nodes can be retained while the remainder of the call-graph is removed; leaf nodes can be removed; *fan-in* functions can be removed (these are functions which are called by a large number of other functions but call no other functions themselves); *fan-out* functions can also be removed (these are functions which call a large number of other functions but may themselves only be called once). These last two operations greatly simplify the call-graphs; sub-graphs can also be removed.

The HTML tool also allows the source code to be viewed. This enables the programmer to select the code from inside the profiling environment.

All these facilities and the resulting call-graphs which they produce can be stored in the profiling history of a program. They can be referred to by the programmer to aid the profiling and code-comprehension process.

6.8 Chapter Summary

The extensions to the cost-centre profiling approach are based on the experience of profiling the LOLITA system, a large real-world system written in Haskell, over a number of years.

The use of cost-centre stacks in profiling is described. This method allows more accurate profiling results to be produced with the use of a post-processor. Inheritance of costs is based on actual calls to functions and not on statistical averaging.

The eval-apply semantics developed by Sansom to describe cost-centre profiling

theory are extended to describe the cost-centre-stack profiling approach. These semantics allow the costs involved in evaluating a Haskell expression to be assigned unambiguously to cost-centre stacks. A new theory of cost inheritance is mathematically defined.

The cost-centre-stack technique is only made possible with an effective implementation. The use of cost-stack tables and cost-stack codes allow codes, rather than larger stacks, to be passed in the execution of a program. The stacks themselves are compressed so that an infinite number of stacks can be addressed in a finite representation. This means that the solution is always feasible, even when profiling large functional systems.

The implementation of the cost-centre-stack profiler required the analysis and re-design of compiler optimisation rules. Transformations in the Core language were producing incorrect cost-centre stacks. New transformation rules were designed to avoid the stacks becoming broken and also to preserve the effectiveness of the optimisations.

A post-processor was implemented using Tk and part of the AMES program maintenance tool. This offers an automated scheme to display the results of profiling in terms of a post-processor and call-graph. The costs of functions can be automatically inherited in this environment. The call-graph can also be automatically simplified and the expensive arms in the call-graph highlighted. A profiling history is kept which allows the programmer to refer to the previous profiling results produced.

Chapter 7

Results and Evaluation

This chapter presents and evaluates some of the results from the cost-centre-stack profiler and the accompanying post-processor. Although the emphasis of the thesis is on the analysis of large-scale systems, the results begin with some smaller example programs. This allows the results of the cost-stack profiler and the cost-centre profiler to be compared for programs which use shared and higher-order functions.

The first example demonstrates the effect which the cost-centre-stack profiler has on the results of shared functions. It is difficult to illustrate the benefits of accurate cost inheritance when working with larger programs; the quantity of code means that it is not always easy to see why the results are so different. Therefore, the first program is only 15 lines long. The results gained from the cost-centre profiler and the cost-centre-stack profiler are significantly different. This first example is also used to explain the post-processing procedure.

The second set of results were collected from the Clausify program, now regarded as one of the profiling benchmarks. Runciman and Wakeling chose the Clausify program to demonstrate the results achieved using their heap profiler. They were able to make substantial improvements to the program. The modified program was later used by Sansom to identify further improvements, not previously discovered using the heap profiler. The program is therefore profiled using the cost-centre-stack profiler and the results are discussed.

Results are then presented from some larger examples. The first is a 10,000 line subset of the LOLITA system which tests the information held in the semantic net of the system. The second set of results is taken from the LOLITA system itself; this extends the results by including a program which contains hundreds of thousands of function calls. The final set of data is collected from the `nofib` benchmark suite supplied by Glasgow University; this allows some general conclusions to be drawn.

An important element in evaluating the results is the consideration of the cost-centre profiler overheads. The success of the profiling scheme is dependent on the conclusion that the overheads of collecting these more detailed results during program execution are acceptable.

7.1 Introductory Example

The first set of results were collected from a simple program, designed to be computationally expensive. The program, which repeatedly reverses lists of numbers, makes use of a number of shared functions. These functions illustrate the difference in the results achieved using a method of cost inheritance, produced by the cost-centre-stack profiler, and a method of flat profiling, produced by the original cost-centre profiler. The example is clearly contrived, but serves to illustrate the basic differences between the two sets of profiling results.

The program

```
> module Main where
> main = print (length a)
> a = (b 1) ++ (c 1)
> b x = (d x) ++ (e x)
> c x = f x
> d x = g (x - 10)
> e x = g x
> f x = (h x) ++ (i x)
> g x = (j [x..100]) ++ (rev (rev (rev (rev [x..100]))))
> h x = j [-1000..100]
> i x = rev (rev (rev (rev [x..100])))
> j 1 = rev (rev (rev (rev (rev (rev 1)))))
> rev = foldl (flip (:)) []        -- A reverse function
```

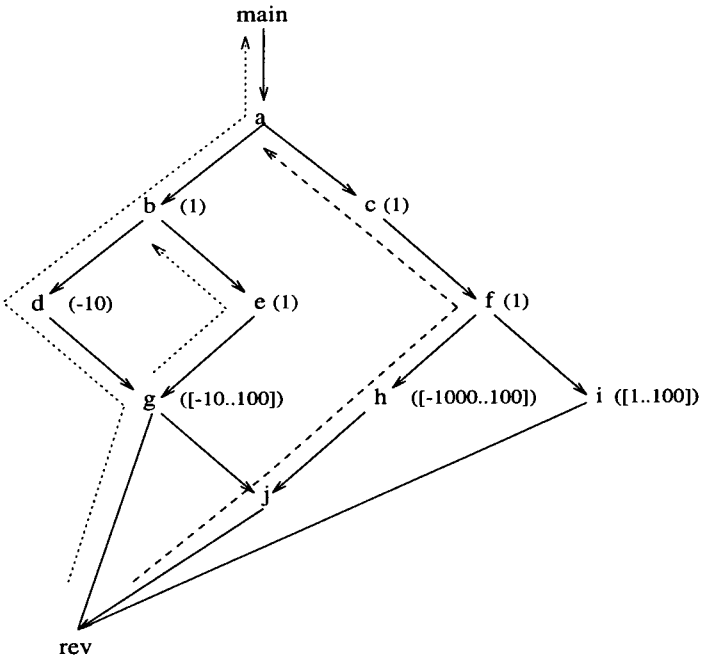


Figure 7.1: Call-graph of experimental program.

is depicted in the call-graph in Figure 7.1. The shared functions g and j serve to illustrate expensive functions, if called with suitably large arguments. It is a function call from h to j which causes the largest amount of computation; the function call from g to j causes significantly less. The arguments passed to the called function are shown in brackets in the figure; for example function e calls function g with the argument 1.

The program is time-profiled with the cost-centre profiler[1] and the results are displayed in Figure 7.2. As expected, the reverse function `rev` (from the program `Main`) accounts for nearly all of the execution time, 99.7% in total[2]. The remaining 0.3% of costs are attributed to the functions `f` (0.1%) and the prelude (0.2%). This last figure is due to the catenation function (++) used throughout the program. The remaining functions show execution costs of 0.0% as they have not registered any time samples during the execution of the program.

[1]Compile-time flags: `-prof -auto-all`; Run-time flags: `-pT`.

[2]The reverse function is included in the Main program to prevent costs being assigned to the PRELUDE library.

```
Thu Jun 20 14:46 1996 Time and Allocation Profiling Report  (Final)
                      (Hybrid Scheme).

           run +RTS -pT -H60M -RTS
```

COST CENTRE	MODULE	GROUP	scc	subcc	%time	%alloc
Main_rev	Main	Main	20867	20834	99.7	99.8
Main_f	Main	Main	3	2	0.1	0.0
Main_a	Main	Main	2	3	0.0	0.0
Main_b	Main	Main	2	2	0.0	0.0
Main_g	Main	Main	2	12	0.0	0.0
MAIN	MAIN	MAIN	1	0	0.0	0.0
Main_c	Main	Main	1	3	0.0	0.0
Main_d	Main	Main	2	1	0.0	0.0
Main_e	Main	Main	1	1	0.0	0.0
Main_h	Main	Main	1	2	0.0	0.0
Main_i	Main	Main	1	4	0.0	0.0
Main_j	Main	Main	6	21	0.0	0.0
PRELUDE	Prelude	Prelude	0	0	0.2	0.1
Main_main_CAF	Main	Main	0	0	0.0	0.0
CAF.Main	Main	Main	0	3	0.0	0.0
Main_h_CAF	Main	Main	0	0	0.0	0.0
Main_g_CAF	Main	Main	0	0	0.0	0.0
Main_i_CAF	Main	Main	0	0	0.0	0.0

Figure 7.2: Results of the cost-centre profiler.

The flat cost-centre profile presented in this example does not provide the programmer with very useful information. It is not clear which of the functions `g`, `j` and `i`, which share the function calls to the utility function `rev`, is responsible for the highest proportion of the costs. Without any re-compilation and re-profiling further results are impossible to calculate.

The cost-centre-stack profiler produces two sets of results. Firstly, a flat profile is produced in the same way as for the cost-centre profiler. This is possible as the implementation maintains the existing *current cost centre*, as well as the *current cost-centre stack*. The flat profile produced by the cost-centre-stack profiler demonstrates that the cost-centre-stack profiler preserves the original flat profile results; this is shown in Figure 7.3. A comparison of these results is important, as

```
Thu Jun 20 15:01 1996 Time and Allocation Profiling Report  (Final)
                      (Hybrid-Cost-Stack Scheme)

           run +RTS -pT -H60M -RTS
```

COST CENTRE	MODULE	GROUP	scc	subcc	%time	%alloc
Main_rev	Main	Main	20867	20834	100.0	100.0
Main_f	Main	Main	3	2	0.0	0.0
Main_a	Main	Main	2	3	0.0	0.0
Main_b	Main	Main	2	2	0.0	0.0
Main_g	Main	Main	2	12	0.0	0.0
MAIN	MAIN	MAIN	1	0	0.0	0.0
Main_c	Main	Main	1	3	0.0	0.0
Main_d	Main	Main	2	1	0.0	0.0
Main_e	Main	Main	1	1	0.0	0.0
Main_h	Main	Main	1	2	0.0	0.0
Main_i	Main	Main	1	4	0.0	0.0
Main_j	Main	Main	6	21	0.0	0.0
PRELUDE	Prelude	Prelude	0	0	0.0	0.0
Main_main_CAF	Main	Main	0	0	0.0	0.0
CAF.Main	Main	Main	0	3	0.0	0.0
Main_h_CAF	Main	Main	0	0	0.0	0.0
Main_g_CAF	Main	Main	0	0	0.0	0.0
Main_i_CAF	Main	Main	0	0	0.0	0.0

Figure 7.3: Results of the cost-centre-stack profiler.

it shows that the results of the original cost-centre profile will not be distorted by the inclusion of the cost-centre stacks in the compiler. The 0.3% error is due to sampling differences.

The cost-centre-stack profiler also produces the following cost-centre-stack results:

```
<Main_rev,Main_j,Main_h,Main_f,Main_c,Main_a,Main_main,> 1181 TICKs
<Main_j,Main_h,Main_f,Main_c,Main_a,Main_main,>             0 TICKs
<Main_f,Main_c,Main_a,Main_main,>                           0 TICKs
<Main_j,Main_g,Main_e,Main_b,Main_a,Main_main,>             0 TICKs
<Main_rev,Main_j,Main_g,Main_e,Main_b,Main_a,Main_main,>   16 TICKs
<Main_g,Main_e,Main_b,Main_a,Main_main,>                    0 TICKs
<Main_rev,Main_g,Main_e,Main_b,Main_a,Main_main,>          10 TICKs
<Main_a,Main_main,>                                         0 TICKs
<Main_b,Main_a,Main_main,>                                  0 TICKs
<Main_i,Main_f,Main_c,Main_a,Main_main,>                    0 TICKs
<Main_rev,Main_i,Main_f,Main_c,Main_a,Main_main,>           7 TICKs
<Main_main,>                                                0 TICKs
<Main_c,Main_a,Main_main,>                                  0 TICKs
<Main_g,Main_d,Main_b,Main_a,Main_main,>                    0 TICKs
<Main_rev,Main_g,Main_d,Main_b,Main_a,Main_main,>          11 TICKs
<Main_j,Main_g,Main_d,Main_b,Main_a,Main_main,>             0 TICKs
<Main_rev,Main_j,Main_g,Main_d,Main_b,Main_a,Main_main,>   12 TICKs
<Main_d,Main_b,Main_a,Main_main,>                           0 TICKs
<Main_e,Main_b,Main_a,Main_main,>                           0 TICKs
<Main_h,Main_f,Main_c,Main_a,Main_main,>                    0 TICKs
Prelude no stack
```

Each cost-centre stack recorded is displayed with the units of time spent computing values within its scope. These results are useful as they immediately indicate the cost stack of the most computationally expensive part of the program. As expected, function `rev` is at the head of this stack:

```
<Main_rev,Main_j,Main_h,Main_f,Main_c,Main_a,Main_main,> 1181 TICKs
```

The function `rev` is also at the head of five further cost-centre stacks:

```
<Main_rev,Main_j,Main_g,Main_e,Main_b,Main_a,Main_main,>   16 TICKs
<Main_rev,Main_g,Main_e,Main_b,Main_a,Main_main,>          10 TICKs
<Main_rev,Main_i,Main_f,Main_c,Main_a,Main_main,>           7 TICKs
<Main_rev,Main_g,Main_d,Main_b,Main_a,Main_main,>          11 TICKs
<Main_rev,Main_j,Main_g,Main_d,Main_b,Main_a,Main_main,>   12 TICKs
```

These show the path of cost centres to the function `rev` via `d`, `e` and `i`. The programmer is presented with a complete set of unambiguous results which avoids there being any misunderstanding when they are interpreted.

The first stage of post-processing involves the cost-centre stacks being transformed, using a C script, into a format which can be interpreted by the graph-tool. The total number of time ticks is calculated for each of the functions at the head of each cost-centre stack. This figure is divided by the total number of time ticks recorded to obtain a percentage. This is equivalent to calculating a flat profile.

```
Total Number of Time Ticks = 1237
```

Cost centre	Ticks as head of CC stack	Total time ticks	%time
Main_rev	1237	1237	100.0
Main_main	0	0	0.0
Main_a	0	0	0.0
Main_b	0	0	0.0
Main_c	0	0	0.0
Main_d	0	0	0.0
Main_e	0	0	0.0
Main_f	0	0	0.0
Main_g	0	0	0.0
Main_h	0	0	0.0
Main_i	0	0	0.0
Main_j	0	0	0.0
Prelude	0	0	0.0

The second stage of post-processing involves the programmer selecting those functions which he is interested in profiling[3]. This activity has been moved from pre-profiling to post-profiling. For the sake of this initial example all functions are selected.

This task is implemented in a second C script, taking the graph-tool input file and producing an augmented input file depending on which cost centres are selected. The resulting file is then loaded into the graph-tool; this is the third stage of post-processing.

[3]Remembering that there is a one to one correspondence between top-level functions and cost centres in this example.

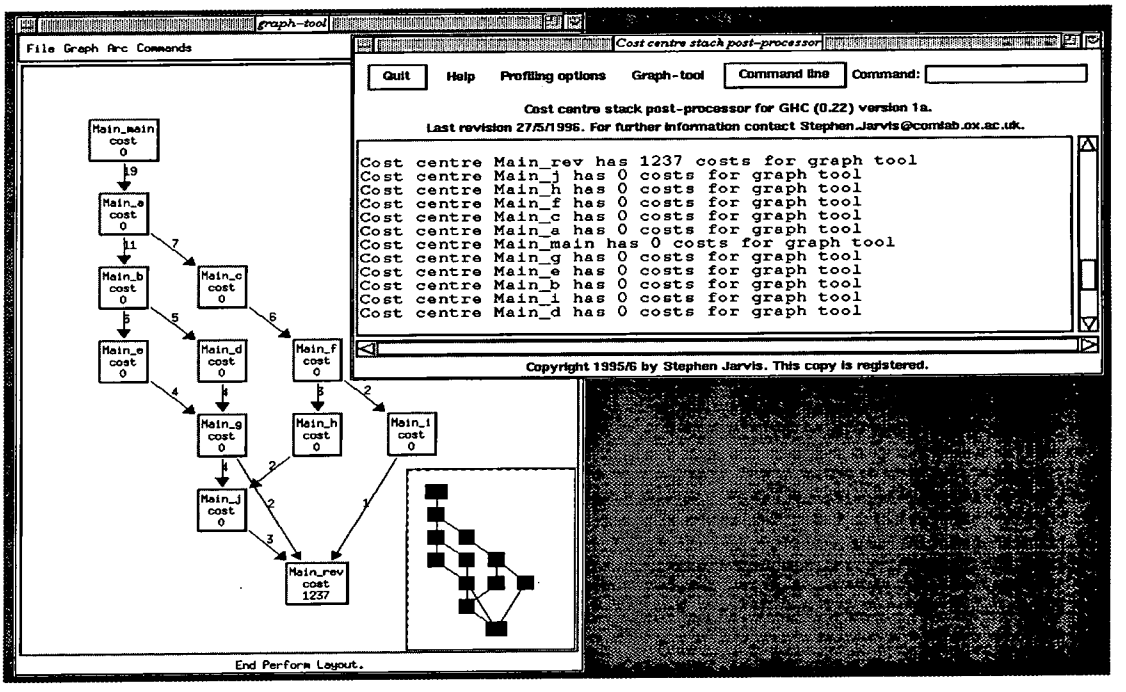


Figure 7.4: Non-inherited post-processed results.

The structure of the program becomes clear when the results are displayed in the graph-tool. In this example the programmer is presented with the call-graph of the program containing all top-level functions. These results are shown in Figure 7.4.

Each node in the graph contains the cost-centre name and the associated costs (in time ticks or as a percentage; the figure shows the former). Each arc in the call-graph[4] is annotated with a number. This number indicates in how many cost-centre stacks this arc was found.

The results are simultaneously displayed in the cost-stack post-processor. The programmer can view the cost-centre stacks, select cost centres or choose different profiling options from this window. All functions are executed within a couple of seconds without any further execution or compilation of the program.

The post-processing tool is also able to perform inheritance of results, accurately inheriting the profiling results to all the selected functions. This is achieved by adding the costs associated with each cost-centre stack to every function in the cost-centre stack. This mechanism is demonstrated below:

[4]An 'arc' connects two nodes.

```
Total Number of Time Ticks = 1237

Cost centre  Ticks in CC stack                  Total time ticks  %time

Main_main    1181 + 16 + 10 + 7 + 11 + 12      1237              100.0
Main_a       1181 + 16 + 10 + 7 + 11 + 12      1237              100.0
Main_b       16 + 10 + 11 + 12                   49                4.0
Main_c       1181 + 7                          1188               96.0
Main_d       11 + 12                             23                1.9
Main_e       16 + 10                             26                2.1
Main_f       1181 + 7                          1188               96.0
Main_g       16 + 10 + 11 + 12                   49                4.0
Main_h       1181                              1181               95.5
Main_i       7                                    7                0.6
Main_j       1181 + 16 + 12                    1209               97.7
Main_rev     1181 + 16 + 10 + 7 + 11 + 12      1237              100.0

Prelude
```

These results can also be displayed by reloading the graph-tool with the new input file, see Figure 7.5. If it was not already clear in the previous results, the expensive arm of the graph now becomes immediately obvious with these inherited results. To emphasise this fact, it is possible to highlight the expensive arm of the graph (or display this arm of the graph only). This may prove to be a useful function in graphs which contain a large number of nodes.

There are two issues which must be addressed in the analysis of these results. The first is the *usefulness* of the cost-centre-stack data and the post-processing techniques for presenting this data. The second is that of the *overheads* involved in collecting this extra data.

7.1.1 Usefulness

The cost-centre stacks allow the construction of call-graphs using dynamic analysis of the actual execution of the program; as each function calls another, a stack of cost centres is constructed showing the sequence of calls to a particular part of the program.

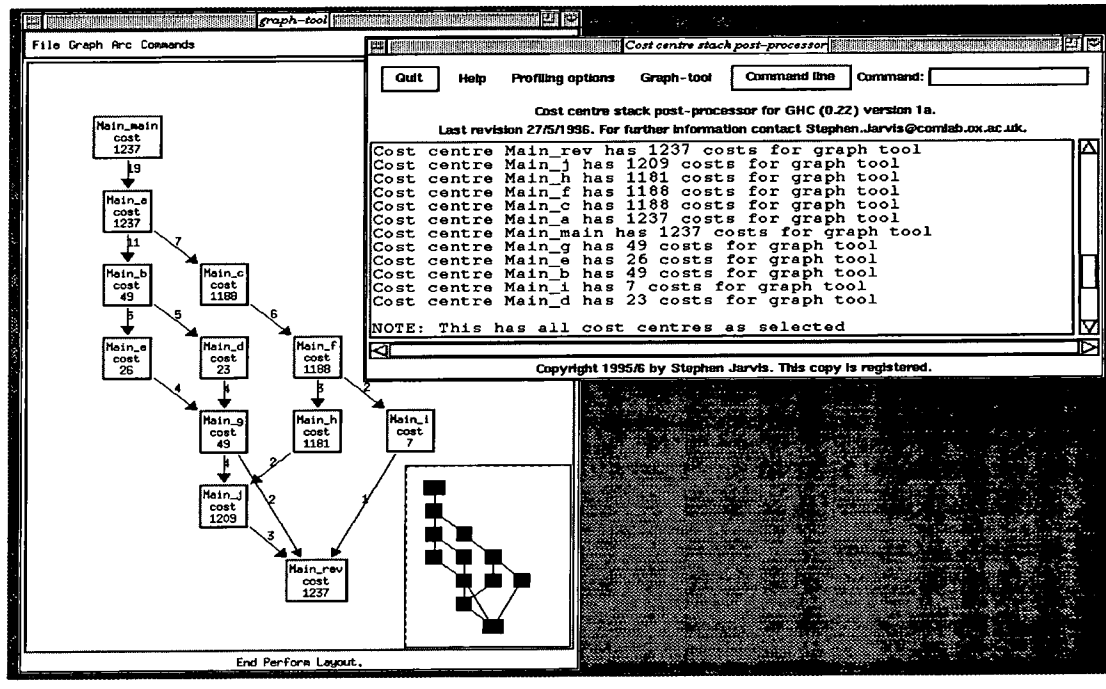


Figure 7.5: Inherited post-processed results.

The cost-centre-stack information allows a call-graph of the program (such as those seen in figures 7.4 and 7.5) to be displayed. Even if the programmer is familiar with the code this makes the task of determining the relationships between parts of the code easier, particularly in the context of a large program. This information has not previously been shown in the profile of a program.

Profiling with cost-centre stacks allows the complete set of program costs to be recorded. They are an accurate record of the program's computational behaviour and therefore a true profile of the program in the sense that no statistical averaging has been used to produce the results.

The post-processor allows these results to be explored instantaneously and without any further execution or compilation of the program. This has not previously been possible when profiling a program. Figure 7.5 shows the complete set of inherited results. These results can be interpreted outside the context of the actual program code; the program graph allows the expensive portion of the code to be identified, leaving the programmer free to identify its cause.

There are a number of profiling and graph-tool options available to the pro-

grammer which allow, amongst other things, the most expensive arm of the graph to be displayed, cost centres to be selected, and flat and inheritance profiles of the program to be produced. None of these options take more than three or four seconds to execute.

The graph in Figure 7.5 clearly shows the distribution of costs in the program, focusing the attention of the programmer on the functions `c`, `f`, `h` and `j`. The programmer can quickly identify the function call from `h` to `j` as causing a significant amount of computation.

Using the post-processor it is also possible to select and de-select cost centres. In the example the programmer might have chosen to view the flat profile with the cost centre `Main_rev` de-selected. The post-processor would accurately subsume the costs of `Main_rev` to its calling functions, again highlighting `Main_j` as part of the expensive arm of the program. Repeating this exercise on `Main_j` would confirm that the call from `Main_h` to `Main_j` was instigating most of the program costs.

A similar top-down (as opposed to bottom-up) approach to profiling can be performed using the post-processor. This is easy to achieve without re-profiling and re-compiling the program and the time benefits are considerable.

7.1.2 Overheads

The study of the overheads for the cost-centre-stack profiling scheme is important. Previous profiling literature has shown that earlier attempts at similar cost-collection methods were abandoned because of the extremely high overheads. The success of the cost-centre-stack method of recording results relies on the fact that the overheads are low enough to make such a system practical.

There are a number of overheads to consider:

- The size of the cost-centre-stack table should not become so large that cost-centre-stack profiling becomes impossible on normal workstations;

- The time that the program takes to compile and any extra heap space needed during compilation should be acceptable. That is, the extra compilation overheads should be small enough to make cost-centre-stack profiling preferable to repeated compilations of a cost-centre profile. Although most of the changes to the Glasgow Haskell Compiler have been to the run-time system, the changes to the compiler optimiser and the generation of extra run-time code are likely to increase the compilation overheads;
- Finally, the execution-time costs should be small enough to make the collection of costs practical. The run-time overheads should not be unacceptably high and the extra heap needed for execution should not be unacceptably large.

These overheads are discussed in turn.

Summary of the cost-centre-stack table

The results of the cost-centre-stack profiler show that 20 cost-centre stacks were produced during the execution of the first example program (when using the `-prof -auto-all` compile time option); there are 12 cost centres in this code.

The structure of the cost-centre-stack table can be displayed by printing the cost-centre stacks and their corresponding index tables. Of these 20 cost-centre stacks the largest was 7 cost centres deep. The index tables of these cost centres were of size 0 (for 1 cost centre), size 1 (for 7 cost centres) and size 2 (for 4 cost centres).

Since each cost-centre-stack table entry stores a pointer to the cost-centre label, a pointer to its index table, pointers to each entry in the index table, their pointers to further stacks and an integer value, the size of the cost-centre-stack table for this example program is 528 bytes. Compared with the total size of the executable which GHC produces, 696320 bytes, these extra bytes are almost insignificant.

Compilation overheads

Heap space needed for compilation:

There is no detectable difference in the size of the heap needed for compilation between the cost-centre profiler and the cost-centre-stack profiler. The GHC default of 4 Megabytes is used.

Time needed to compile the program:

The cost-centre profiler takes a total of 94.3 seconds to compile and link the program. This time is taken from an average of ten compilations using the unix system time command.

Using the same method of analysis, the cost-centre-stack profiler takes a total of 98.5 seconds to compile and link the code. This gives a time overhead of 5.57%. These compilation overheads are acceptable.

Executable differences

Size of the two executable files:

The cost-centre profiler produces an executable file of 696320 bytes. The cost-centre-stack profiler produces an executable file of 712704 bytes. This gives a 2.35 % size overhead when using the cost-centre-stack profiler.

Run-time speed and memory usage:

The execution time measured using the unix time command is averaged over 10 executions of the program. The cost-centre profiler produces an executable which runs in 130.5 seconds; the cost-centre-stack profiler runs in 133.5 seconds. These run-time overheads are 2.29 %. The size of the heap needed to execute the program is the same for both profilers.

These results are encouraging for small programs, but the results of some larger programs must also be considered.

7.2 The Clausify Program

The Clausify program (found in Appendix A) has emerged as a classic benchmark in the analysis of profiling tools. Runciman and Wakeling used this example to demonstrate the results of their heap profiling tool; they also produced improvements to the code by reducing the memory needed for the execution of the program from 1.3Mb to 7Kb. Similarly Sansom profiled the resulting program from Runciman and Wakeling and improved the code by a further 25 percent. This study is extended by profiling the program with the cost-centre-stack profiler.

The Clausify program takes a series of propositional formulae as its input and produces the clausal form equivalents as its results[5]. The transformation of each proposition to a set of clauses is specified by the following rules:

- *elim* eliminates equivalence and implications:

  $p = q \rightarrow (p \Rightarrow q) \wedge (q \Rightarrow p)$

  $p \Rightarrow q \rightarrow \neg p \vee q$

- *negin* makes negations the innermost connectives:

  $\neg\neg p \rightarrow p$

  $\neg(p \vee q) \rightarrow \neg p \wedge \neg q$

  $\neg(p \wedge q) \rightarrow \neg p \vee \neg q$

- *disin* pushes disjuncts within conjuncts:

  $p \vee (q \wedge r) \rightarrow (p \vee q) \wedge (p \vee r)$

  $(p \vee r) \wedge r \rightarrow (p \vee r) \wedge (q \vee r)$

- The function *split* splits up the conjuncts:

  $p \wedge q \rightarrow p$

  q

[5]This description is based on the original one by Runciman and Wakeling [Runciman and Wakeling, 1993].

- The function *unicl* forms a set of unique non-tautologous clauses:

$$p_1 \vee ... \vee p_n \vee \neg q_1 \vee ... \vee \neg q_m \quad \rightarrow \quad (\{p_1, ..., p_n\}, \{q_1, ..., q_m\})$$

A clause (*ps*,*qs*) is tautologous if $(ps \cap qs) \neq \emptyset$

The propositional formulae are represented by the following data type:

```
data Formula = Sym Char
             | Not Formula
             | Dis Formula Formula
             | Con Formula Formula
             | Imp Formula Formula
             | Eqv Formula Formula
```

The transformation rules are implemented with this data structure. The rules are combined using the following pipeline:

```
clauses = unicl . split . disin . negin . elim
```

7.2.1 A cost-centre profile

The Clausify program has already been used as a benchmark experiment for the cost-centre profiler. A time profile, similar to those seen in [Sansom, 1994], is shown in Figure 7.6.

These results are collected using the `-auto-all` run-time flag, so that all top-level functions in the code are annotated with a cost centre.

A hybrid profiling scheme is used which allows the costs of dictionary or library functions and the costs produced by CAFs to be subsumed by the function in which they are lexically defined. This is useful in this example because a lexical profile of the program produces a cost centre `unicl` with zero costs attributed to it.

This is because the function

```
unicl = filterset (not . tautclause) . map clause
```

is a CAF. A lexical scheme of cost attribution assigns the costs of applying this function to a CAF introduced by the compiler. In a similar way, the costs accrued by library or dictionary functions are also attributed to CAFs which are used to construct the dictionary. The hybrid profiling scheme was developed by Sansom in response to the difficulties experienced in deciphering these CAF costs.

In Sansom's thesis, the results of profiling the Clausify program are analysed. To allow the profiling results to be compared, Sansom adopts the same input data as Runciman and Wakeling. The proposition

$$(a = a = a) \;=\; (a = a = a) \;=\; (a = a = a)$$

reduces to the single clause $(a, \emptyset)$, although a substantial amount of work is produced in the process.

Sansom identified time spent in the `clause`, `insert` and `tautclause` functions as being a significant problem with the code. Much of this time was due to the throwing away of the clauses and duplicate symbols during execution.

Using this analysis, Sansom was able to rewrite the program using unboxed characters, a method of code optimisation built into the Glasgow Haskell Compiler. The removal of the clauses and duplicate symbols relied on comparing characters. By introducing unboxed characters, the program was improved by 25%.

7.2.2 A cost-centre-stack profile

To produce a cost-centre-stack profile of the Clausify program, the program is also profiled with the `-auto-all` run-time profiling flag. The non-zero cost-centre-stack results are shown in Figure 7.7.

COST CENTRE	MODULE	GROUP	scc	subcc	%time	%alloc
insert	Main	Main	52398	0	20.9	0.0
clause	Main	Main	110142	157194	18.6	51.6
CAF:unicl	Main	Main	5347	10693	12.8	15.8
disin'	Main	Main	12214	12164	9.3	12.0
tautclause	Main	Main	5346	0	5.8	8.8
Main_filterset	Main	Main	2	5347	5.8	0.0
split	Main	Main	10692	10691	4.7	5.3
filterset	Main	Main	1	5346	3.5	0.0
Main_insert	Main	Main	52398	52398	3.5	0.0
filterset'	Main	Main	5347	0	2.3	5.3
negin	Main	Main	231	230	2.3	0.3
elim	Main	Main	199	198	0.0	0.3
MAIN	MAIN	MAIN	1	1	0.0	0.2
disin	Main	Main	199	248	0.0	0.2
clausify	Main	Main	2	3	0.0	0.2
parse'	Main	Main	28	43	0.0	0.0
while	Main	Main	12	20	0.0	0.0
red	Main	Main	8	0	0.0	0.0
CAF:clausifyline	Main	Main	2	4	0.0	0.0
interleave	Main	Main	9	6	0.0	0.0
CAF:main	Main	Main	1	2	0.0	0.0
disp	Main	Main	1	2	0.0	0.0
parse	Main	Main	1	1	0.0	0.0
CAF:clauses	Main	Main	2	5	0.0	0.0
CAF:redstar	Main	Main	12	12	0.0	0.0
Main_filterset	Main	Main	1	1	0.0	0.0
Main_opri	Main	Main	26	26	0.0	0.0
Main_spri	Main	Main	20	20	0.0	0.0
Main_tautclause	Main	Main	5346	5346	0.0	0.0
opri	Main	Main	52	26	0.0	0.0
spri	Main	Main	22	20	0.0	0.0
CAF.Main	Main	Main	0	2	10.5	0.0
PRELUDE	Prelude	Prelude	0	0	0.0	0.0
CAF:main_CAF	Main	Main	0	0	0.0	0.0
CAF:redstar_CAF	Main	Main	0	12	0.0	0.0
CAF:spaces_CAF	Main	Main	0	0	0.0	0.0
disp_CAF	Main	Main	0	0	0.0	0.0
parse'_CAF	Main	Main	0	0	0.0	0.0
clause_CAF	Main	Main	0	0	0.0	0.0
filterset_CAF	Main	Main	0	0	0.0	0.0
parse_CAF	Main	Main	0	0	0.0	0.0
split_CAF	Main	Main	0	0	0.0	0.0

Figure 7.6: Results of the cost-centre profiler.

```
<insert,Main_insert,Main_clause,CAF:unicl,Main_clauses,
 CAF:clausifyline,Main_clausify,Main_main,>                    130
<clause,CAF:unicl,Main_clauses,CAF:clausifyline,
 Main_clausify,Main_main,>                                     126
<CAF:unicl,Main_clauses,CAF:clausifyline,Main_clausify,
 Main_main,>                                                    88
<CAF.Main,>                                                     60
<Main_filterset,filterset,CAF:unicl,Main_clauses,
 CAF:clausifyline,Main_clausify,Main_main,>                     44
<disin',disin,Main_clauses,CAF:clausifyline,Main_clausify,
 Main_main,>                                                    43
<Main_insert,clause,CAF:unicl,Main_clauses,Main_clausifyline,
 Main_clausify,Main_main,>                                      35
<split,Main_clauses,CAF:clausifyline,Main_clausify,
 Main_main,>                                                    32
<negin,Main_clauses,CAF:clausifyline,Main_clausify,
 Main_main,>                                                    16
<CAF:clausifyline,Main_clausify,Main_main,>                     16
<tautclause,Main_tautclause,CAF:unicl,Main_clauses,
 CAF:clausifyline,Main_clausify,Main_main,>                     15
<Main_clausifyline,Main_clausify,Main_main,>                    15
<filterset,Main_filterset,CAF:unicl,Main_clauses,
 CAF:clausifyline,Main_clausify,Main_main,>                     11
<Main_tautclause,CAF:unicl,Main_clauses,Main_clausifyline,
 Main_clausify,Main_main,>                                       7
<Main_insert,insert,Main_clause,CAF:unicl,Main_clauses,
 CAF:clausifyline,Main_clausify,Main_main,>                      2
<parse',parse,CAF:clausifyline,Main_clausify,Main_main,>         1
<split,Main_clauses,CAF:clausifyline,Main_clausify,
 Main_main,>                                                     1
<Main_insert,insert,Main_clause,CAF:unicl,Main_clauses,
 CAF:clausifyline,Main_clausify,Main_main,>                      1
Prelude no stack
```

Figure 7.7: Non-zero cost-centre-stack results.

A flat profile is produced at the first stage of post-processing. These results can again be compared with the flat profile produced by the cost-centre profiler. The flat profile is summarised to include only those cost centres which have non-zero costs.

```
Total Number of Time Ticks = 643
```

Cost centre	Ticks as head of CC stack	Total time ticks	%time
Main_insert	35 + 2 + 1	38	5.9
CAF.Main	60	60	9.4
parse'	1	1	0.2
split	1 + 32	33	5.1
clause	126	126	19.6
insert	130	130	20.2
Main_filterset	44	44	6.8
CAF:clausifyline	16	16	2.5
Main_clausifyline	15	15	2.3
disin'	43	43	6.7
filterset	11	11	1.7
CAF:unicl	88	88	13.7
Main_tautclause	7	7	1.1
tautclause	15	15	2.3
negin	16	16	2.5

These results are displayed in full in Figure 7.8 [6].

The non-inherited results can be inherited using the post-processing tool. Again, for the sake of this example, all top-level functions are deemed to be cost centres. Figure 7.9 summarises the inheritance process for all the cost-centre stacks with non-zero costs.

The cost-centre stacks are loaded into the post-processor and the graph-tool is invoked displaying only those stacks with non-zero costs[7]. The post-processor display for the non-inherited results is found in Figure 7.10. These results can then

[6]References to `insert` and `Main_insert` should be combined as they refer to the same part of the source code.

[7]Some cost centres will not therefore appear in the graph, so that the programmer can focus his attention on only those parts of the program which register some costs.

COST CENTRE	MODULE	GROUP	scc	subcc	%time
insert	Main	Main	52369	0	20.2
clause	Main	Main	110142	157194	19.6
CAF:unicl	Main	Main	5347	10693	13.7
CAF.Main	Main	Main	0	2	9.4
Main_filterset	Main	Main	2	5347	6.8
disin'	Main	Main	12214	12164	6.7
Main_insert	Main	Main	52398	52398	5.9
split	Main	Main	10692	10691	5.1
CAF:clausifyline	Main	Main	2	4	2.5
negin	Main	Main	231	230	2.5
Main_clausifyline	Main	Main	2	4	2.3
tautclause	Main	Main	5346	0	2.3
filterset	Main	Main	1	5346	1.7
Main_tautclause	Main	Main	5346	5346	1.1
parse'	Main	Main	28	43	0.2
Main_spri	Main	Main	20	20	0.0
Main_opri	Main	Main	26	26	0.0
Main_insert	Main	Main	52398	52398	0.0
spri	Main	Main	22	20	0.0
red	Main	Main	8	0	0.0
while	Main	Main	12	20	0.0
opri	Main	Main	52	26	0.0
parse	Main	Main	1	1	0.0
elim	Main	Main	199	198	0.0
interleave	Main	Main	9	6	0.0
disin	Main	Main	199	248	0.0
CAF:clauses	Main	Main	2	5	0.0
clausify	Main	Main	2	3	0.0
filterset'	Main	Main	5347	0	0.0
Main_filterset	Main	Main	1	1	0.0
CAF:redstar	Main	Main	12	12	0.0
Prelude no stack	Prelude	(null)			

Figure 7.8: Flat profile results of the cost-centre-stack profiler.

be automatically inherited to all the cost centres in the call-graph, giving overall inherited results for the program; Figure 7.11.

Sansom identified that time spent in the `clause`, `insert` and `tautclause` functions accounted for a large amount of the time spent executing the program, due to clauses and duplicate symbols being thrown away.

The cost-centre-stack results indicate similar behaviour. The non-inherited results show that the functions `insert` and `clause` are together responsible for 40% of the total execution time. The cost-centre profile does not immediately suggest that the function `tautclause` might be amongst those undertaking extra work, and Sansom's conclusions may have been based on careful analysis of the program as well as the profiling results.

The non-inherited cost-centre-stack results also show that the function `unicl` is responsible for a large proportion of the total costs. It is therefore the functions `CAF:unicl`, `insert` and `clause` on which the programmer's attention will initially be focused.

The inherited results extend these observations by showing a bottleneck at the `CAF:unicl` cost centre; after this point in the graph all the inherited costs are large.

There are two observations which can be made about inherited results:

- If a cost centre has a small number of inherited costs and is itself inexpensive then there is no way that any change to this function will help to improve the program This is not true of the cost-centre profiler where cost centres with low or even zero costs may still contain a performance bug;
- Conversely, if a cost centre has a large inherited cost and is itself inexpensive then it may well be worth some attention, as this function may be the cause of large costs lower down in the program graph. This is different to the way in which the programmer would interpret a flat profile of the program.

The cost centre `Main_clauses` satisfies the second of these two observations therefore the programmer's attention can be focused on the `clauses` pipeline.

Cost centre	Ticks in CC stack	Total time ticks	%time
Main_main	130 + 126 + 88 + 44 + 1 + 43 + 35 + 32 + 16 + 16 + 15 + 15 + 11 + 7 + 2 + 1 + 1	583	90.7
Main_clausify	130 + 126 + 88 + 44 + 43 + 35 + 32 + 16 + 16 + 15 + 15 + 11 + 7 + 2 + 1 + 1 + 1	583	90.7
CAF:clausifyline	130 + 126 + 88 + 44 + 43 + 35 + 32 + 16 + 16 + 15 + 11 + 7 + 2 + 1 + 1 + 1	568	88.3
Main_clauses	130 + 126 + 88 + 44 + 43 + 35 + 32 + 16 + 15 + 11 + 7 + 2 + 1 + 1	551	85.7
CAF:unicl	130 + 126 + 88 + 44 + 35 + 15 + 11 + 7 + 2 + 1	459	71.4
clause	126 + 35	161	25.0
Main_insert	130 + 35 + 2 + 1	168	26.1
Main_clause	130 + 2 + 1	133	20.7
insert	130 + 2 + 1	133	20.7
CAF.Main	60	60	9.4
Main_filterset	44 + 11	55	8.6
filterset	44 + 11	55	8.6
disin'	43	43	6.7
disin	43	43	6.7
split	32 + 1	33	5.1
Main_tautclause	15 + 7	22	3.4
negin	16	16	2.5
tautclause	15	15	2.3
Main_clausifyline	15	15	2.3
parse'	1	1	0.2
parse	1	1	0.2

Figure 7.9: Table summarising the inheritance of the Clausify results.

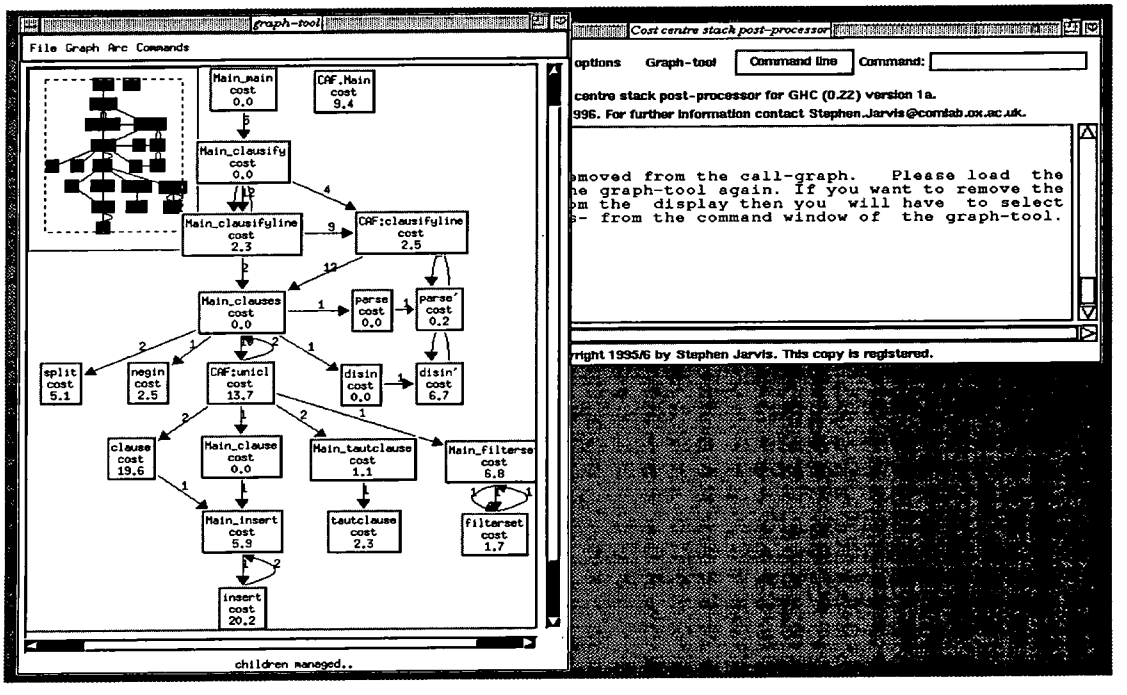


Figure 7.10: Non-inherited Clausify results from the cost-centre-stack profiler.

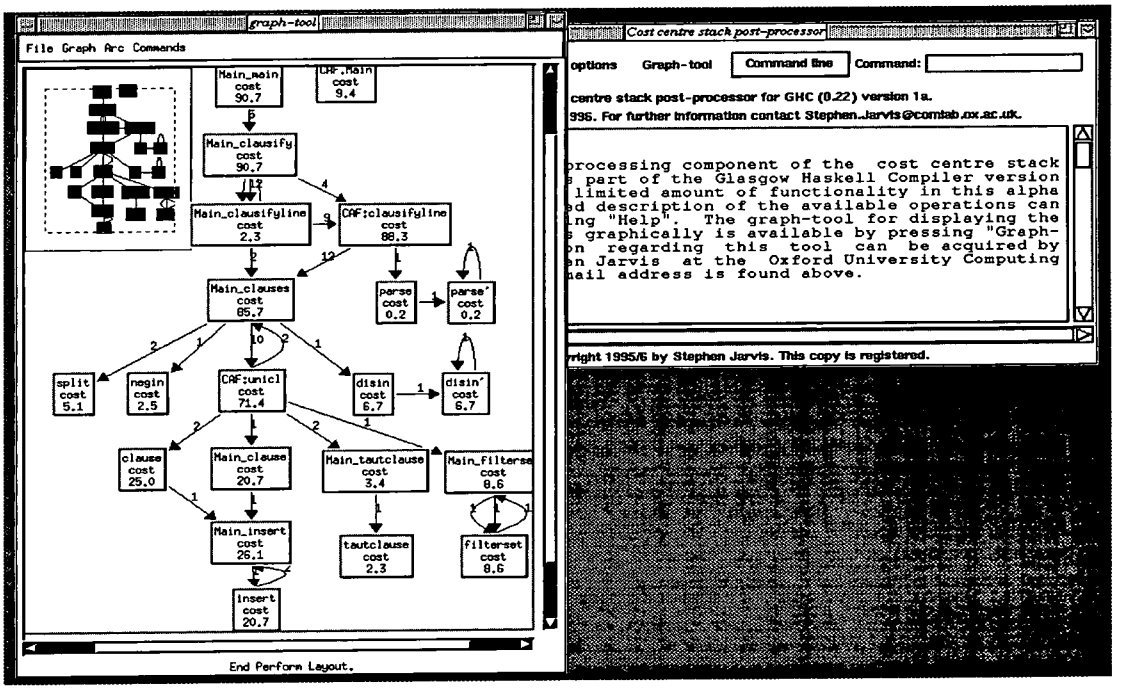


Figure 7.11: Inherited Clausify results from the cost-centre-stack profiler.

`CAF:unicl` also has a large inherited cost so this is also considered in the analysis of the program.

It should be noted that, since there are no problems with shared functions in this program, the inheritance observations (above) are used in the analysis of the program costs.

Analysis of the profiling results would suggest that something in the pipeline

```
clauses = unicl . split . disin . negin . elim
```

is causing `clauses` to have huge inherited costs. It is also causing `CAF:unicl` to be expensive in its own right, therefore indicating that it might be before this point in the pipeline that the problem occurs. Analysis of the program is now simple, the programmer just tests to see what result is produced by each function in the pipeline. The results of the pipeline are demonstrated with a Gofer version of the Clausify program.

A parse of the propositional formulae, `(a=a=a)=(a=a=a)=(a=a=a)`,

```
parse "(a = a = a) = (a = a = a) = (a = a = a)"
```

produces the following expression:

```
Eqv (Eqv (Sym 'a') (Eqv (Sym 'a') (Sym 'a')))
    (Eqv (Eqv (Sym 'a') (Eqv (Sym 'a') (Sym 'a')))
    (Eqv (Sym 'a') (Eqv (Sym 'a') (Sym 'a')))) :: Formula
(415 reductions, 770 cells)
```

Applying the `elim` function to this result

```
(elim.parse) "(a = a = a) = (a = a = a) = (a = a = a)"
```

to remove the connectives, other than $\neg$, $\vee$ and $\wedge$, results in a formula requiring the following amount of computation:

```
(1150 reductions, 3884 cells)
```

Note the number of reductions needed for this operation. Applying the `disin` function, which moves disjunctions within conjunctions, explodes this formula to output which is enormous! The Gofer session produces the following execution statistics which indicate the size of such a result.

```
(220603 reductions, 1472673 cells, 28 garbage collections)
```

As a consequence of this, the `unicl` part of the program becomes expensive and the `insert` function (via `clause`), which is used in `CAF:unicl`, is also expensive.

This problem can be fixed in a number of ways. Sansom identified a problem with the `clause`, `insert` and `tautclause` functions and produced a fix using unboxed integers to reduce the comparison operation during list insert. This produced improvements of 25% to the code. It is noted that this is a low-level improvement which may not be immediately obvious to an applications programmer.

Alternatively the fix can be implemented at a higher level, for example in the actual algorithm of the program. Using the cost-centre-stack profiler to identify the general problem with the size of the formula being produced, a more comprehensive fix was sought which would reduce the size of the formula being produced.

The point at which this program starts to become expensive is after the call to the `disin` function, a function which is moderately expensive itself, but more importantly produces a result which causes the rest of the program after this function call to be hugely expensive.

Runciman and Röjemo were also able to identify this part of the program as expensive with their retainer heap profiler [Runciman and Röjemo, 1996]. The fix which they proposed to reduce the heap space used during program execution is used in this context to reduce the time taken during program execution.

Runciman and Röjemo's proposed program fix was based on the idea that, rather than `disin` reconstructing the disjunctive trees (since they were found in its

argument), it would normalise them so that some pruning could take place earlier in the pipeline; this, they state, is known as *filter promotion.* These improvements were transferred to the program in terms of: a normalising constructor `dis` which is used in place of `Dis`; a function to compare literals, `complit`; and an extra `Formula` constructor `Tru`. The two versions of the Clausify program are included in Appendix A.

The changes to the Clausify program reduce the execution time of the program from 14.80 seconds to 0.04 seconds; this is an improvement of 370%.

7.2.3 Usefulness

The results from the cost-centre-stack profiler offer the programmer two alternative ways of addressing the program's efficiency:

- Firstly, the flat profile indicates in which of the functions most of the program's execution time is spent. This gives the programmer the option of implementing low-level improvements to those parts of the code, which will in turn improve the overall execution time;
- Secondly, the inheritance profile, which has the ability to inherit costs to functions higher in the call-graph, may cause the programmer to look at higher-level changes, perhaps to the algorithm of the program. It is the time needed to perform this task which is greatly reduced by the cost-centre-stack profiler.

The Clausify program provides an interesting example as the changes which are finally made to the code are to a function which in the flat profile appears to be inexpensive. Even in the inheritance profile of the program the `disin` function does not appear to have a high cost. This is of course true, yet it is not the costs of the `disin` function which are a problem in the code, but the results produced by this function which make the calculations in the rest of the program expensive. The advantage of the cost-centre-stack profiler is that it gives the programmer an

environment in which to explore these results and offers the context in which to consider high- or low-level possibilities for program improvements.

It is important to note that the benefits of the cost-centre-stack profiler, and in particular the post-processor, are not just limited to programs which make extensive use of shared functions. This example demonstrates this fact.

The cost-centre-stack profiling of the Clausify program was based solely on time profiling. These results have not previously been shown with a time profile; Runciman and Röjemo demonstrated similar results with a heap retainer profiler. This is also the first time that these results have been collected using a profiler based on the GHC compiler.

The cost-centre-stack profiler clearly provides more of an insight into the program's behaviour than the cost-centre profiler. Sansom's results can be replicated, demonstrating results which were not revealed by the Runciman and Wakeling heap profiler. The post-processor allows the programmer to analyse the program rapidly and identify problems in the algorithm permitting more than just low-level fixes to the code. The code changes proposed by Runciman and Röjemo were used to demonstrate huge improvements to the speed of the program in response to this time problem. From this example, the cost-centre-stack profiling tool appears to be very versatile.

7.2.4 Overheads

Compilation overheads

Heap space needed for compilation:

There are no differences in the size of the heap needed for compilation between the cost-centre profiler and the cost-centre-stack profiler. 4Mb of heap is specified during compilation for both profilers.

Time needed to compile the program:

The cost-centre profiler takes a total of 132.7 seconds to compile and link the program. This time is taken from an average of ten compilations using the unix system time command.

The cost-centre-stack profiler, using the same method of analysis, takes a total of 154.4 seconds to compile and link the code. This gives a 16.6% overhead when compiling with the cost-centre-stack profiler as opposed to the cost-centre profiler.

Executable differences

Size of the two executable files:

The cost-centre profiler produces an executable file of 745472 bytes. The cost-centre-stack profiler produces an executable file of 770048 bytes. This gives a 3.3% space overhead when using the cost-centre-stack profiler.

Run-time speed and memory usage:

Both the cost-centre profiler and the cost-centre-stack profiler will execute the Clausify program in 4Mb of heap.

The time overheads for the execution of the program are 33.3% based on the 26 cost centres which are produced with the `-auto-all` compile flag. This is a higher run-time overhead than those found during the execution of smaller example programs. For the cost-centre-stack profiler to be successful, it is important that there is the same proportion of overheads for large programs. The remainder of the case studies in this chapter are dedicated to large programs. The analysis and evaluation focus on the overheads involved, in order to demonstrate the suitability of the scheme for large functional programs.

7.3 ntest— A LOLITA subset

`ntest` is a utility program which allows LOLITA programmers to interrogate nodes in the LOLITA semantic network. There are two basic operations which the `ntest` program provides. The first is a facility which allows the programmer to enter a node number, resulting in the concept stored at that node in the net; the second allows the programmer to enter a concept, which will be searched for in the semantic network, resulting in the node number at which this concept is stored.

The program makes use of a substantial portion of the utility and support modules of the LOLITA system. The modules included in the program contain 4036 lines of Haskell code and 6571 lines of C code, a total of 10607 lines of code. The program also generates a further 3500 lines of C code automatically. It is therefore a useful test program for the lower levels of the LOLITA system; it is also a substantial test case on which to test the cost-centre-stack profiler.

7.3.1 ntest results

Profiling the ntest program with the cost-centre profiler produces the results shown in Figure 7.12. Analysis of these results shows that loading the semantic-net structure accounts for 97.2% of the total execution time; interrogating the semantic network accounts for only a small percentage of the overall execution time.

Partial results from the cost-centre-stack profiler are shown in Figure 7.13. These results correspond to the flat cost-centre profile, although sampling differences have given slightly more interesting results.

Sat Dec 16 19:24 1995 Time and Allocation Profiling Report (Final)
(Hybrid Scheme)

COST CENTRE	MODULE	GROUP	scc	subcc	%time	%alloc
StaticNet_loadZu	StaticNet	StaticNet	1	1	97.2	12.8
Main_doIt	Main	Main	2	10	2.8	20.2
MAIN	MAIN	MAIN	1	1	0.0	27.4
ContIprog_writeC	ContIprog	ContIprog	5	0	0.0	4.3
ContIprog_readln	ContIprog	ContIprog	4	2	0.0	2.1
ContIprog_readln	ContIprog	ContIprog	6	7	0.0	1.8
Main_showSize	Main	Main	2	2	0.0	1.5
StaticNet_mkIntL	StaticNet	StaticNet	16	11	0.0	1.2
StaticNet_ghcBui	StaticNet	StaticNet	10	5	0.0	0.9
StaticNet_badGen	StaticNet	StaticNet	5	10	0.0	0.8
StaticNet_sNodeL	StaticNet	StaticNet	1	2	0.0	0.6
ContIprog_runCIP	ContIprog	ContIprog	1	0	0.0	0.5
StaticNet_sNodeA	StaticNet	StaticNet	6	6	0.0	0.4
ContIprog_getPro	ContIprog	ContIprog	2	1	0.0	0.4
StaticNet_sNodeA	StaticNet	StaticNet	1	7	0.0	0.4
Main_main	Main	Main	4	10	0.0	0.3
Main_interactive	Main	Main	5	8	0.0	0.3
Stdenv2_takeZuex	Stdenv2	Stdenv2	3	0	0.0	0.3
Stdenv2_dropZuex	Stdenv2	Stdenv2	2	1	0.0	0.2
StaticNet_sNodeC	StaticNet	StaticNet	1	3	0.0	0.2
StaticNet_sNodeS	StaticNet	StaticNet	1	2	0.0	0.2
StaticNet_sNetSi	StaticNet	StaticNet	1	1	0.0	0.2
StaticNet_arcsOf	StaticNet	StaticNet	2	2	0.0	0.1
StaticNet_arcsOf	StaticNet	StaticNet	1	3	0.0	0.1
ContIprog_endCIP	ContIprog	ContIprog	1	0	0.0	0.1
Main_loadIt	Main	Main	1	2	0.0	0.1
StaticNet_mkstri	StaticNet	StaticNet	2	1	0.0	0.1
ContIprog_selDia	ContIprog	ContIprog	6	1	0.0	0.0
StaticNet_cvtBoo	StaticNet	StaticNet	5	0	0.0	0.0
StaticNet_genera	StaticNet	StaticNet	1	0	0.0	0.0
StaticNet_hackIn	StaticNet	StaticNet	5	0	0.0	0.0
StaticNet_isGene	StaticNet	StaticNet	1	1	0.0	0.0
StaticNet_mkNode	StaticNet	StaticNet	5	5	0.0	0.0
StaticNet_sIniti	StaticNet	StaticNet	1	0	0.0	0.0
StaticNet_sNetLo	StaticNet	StaticNet	1	1	0.0	0.0
Stdenv2_sTail	Stdenv2	Stdenv2	1	0	0.0	0.0
Stdenv2_stl	Stdenv2	Stdenv2	1	1	0.0	0.0
PRELUDE	Prelude	Prelude	0	0	0.0	16.0
CAF.Main	Main	Main	0	1	0.0	4.0
CAF.StaticNet	StaticNet	StaticNet	0	0	0.0	1.9
CAF.ContIprog	ContIprog	ContIprog	0	5	0.0	0.3
ContIprog_readln	ContIprog	ContIprog	0	0	0.0	0.2
Main_doIt_CAF	Main	Main	0	0	0.0	0.2
Main_showSize_CA	Main	Main	0	0	0.0	0.2
ContIprog_endCIP	ContIprog	ContIprog	0	0	0.0	0.1
ContIprog_runCIP	ContIprog	ContIprog	0	0	0.0	0.1
Main_loadIt_CAF	Main	Main	0	0	0.0	0.1

Figure 7.12: Results of the cost-centre profiler.

```
<StaticNet_load_Binary,StaticNet_sNetLoad,Main_loadIt,
 Main_main,>                                                  with    54 TICKs
<StaticNet_badGeneration,StaticNet_sNodeArcTargets,
 StaticNet_ghcNodeLinks,StaticNet_sNodeLinks,Main_doIt,
 Main_interactive,Main_loadIt,Main_main,>                     with     2 TICKs
<StaticNet_isGenerationError,StaticNet_load_Binary,
 StaticNet_sNetLoad,Main_loadIt,Main_main,>                   with     1 TICKs
<StaticNet_hackInitialiseNumb,StaticNet_badGeneration,
 StaticNet_sNetSize,Main_showSize,Main_loadIt,
 Main_main,>                                                  with     1 TICKs
<StaticNet_badGeneration,StaticNet_sNodeString,Main_doIt,
 Main_interactive,Main_loadIt,Main_main,>                     with     1 TICKs
<StaticNet_mkstringZh,StaticNet_sNodeString,Main_doIt,
 Main_interactive,Main_loadIt,Main_main,>                     with     1 TICKs
<Stdenv2_take_exclude,Main_doIt,Main_interactive,
 Main_loadIt,Main_main,>                                      with     1 TICKs
<Main_doIt,Main_interactive,Main_main,>                       with     0 TICKs
<ContIprog_writeCIP,Main_showSize,Main_loadIt,Main_main,>with     0 TICKs
<ContIprog_readlnECIP,ContIprog_selDial,ContIprog_readlnCIP,
 ContIprog_getProgArgsCIP,ContIprog_runCIP,Main_main,>        with     0 TICKs
<ContIprog_readlnCIP,ContIprog_selDial,ContIprog_getProgArgsCIP,
 ContIprog_runCIP,Main_main,>                                 with     0 TICKs
<Main_showSize,Main_loadIt,Main_main,>                        with     0 TICKs
<StaticNet_generationError,StaticNet_isGenerationError,
 StaticNet_load_Binary,StaticNet_sNetLoad,Main_loadIt,
 Main_main,>                                                  with     0 TICKs
<StaticNet_sNetLoad,Main_loadIt,Main_main,>                   with     0 TICKs
<Main_main,>                                                  with     0 TICKs
<StaticNet_sInitialiseData,Main_loadIt,Main_main,>            with     0 TICKs
<ContIprog_readlnCIP,Stdenv2_take_exclude,Main_doIt,
 Main_interactive,Main_loadIt,Main_main,>                     with     0 TICKs
<ContIprog_selDial,ContIprog_readlnCIP,Stdenv2_take_exclude,
 Main_doIt,Main_interactive,Main_loadIt,Main_main,>           with     0 TICKs
<ContIprog_endCIP,ContIprog_selDial,ContIprog_readlnCIP,
 Stdenv2_take_exclude,Main_doIt,Main_interactive,
 Main_loadIt,Main_main,>                                      with     0 TICKs
<ContIprog_selDial,Stdenv2_take_exclude,Main_doIt,
 Main_interactive,Main_loadIt,Main_main,>                     with     0 TICKs
```

Figure 7.13: Partial results of the cost-centre-stack profiler.

In summary the non-zero, non-inherited results from the cost-centre-stack profiler are:

COST CENTRE	MODULE	GROUP	scc	subcc	%time
StaticNet_load_Binary	StaticNet	StaticNet	1	1	88.52
StaticNet_badGeneration	StaticNet	StaticNet	5	10	4.83
StaticNet_isGenerationError	StaticNet	StaticNet	1	1	1.61
StaticNet_hackInitialiseNumb	StaticNet	StaticNet	5	0	1.61
StaticNet_mkstringZh	StaticNet	StaticNet	2	1	1.61
Stdenv2_takeZuexclude	StaticNet	StaticNet	3	0	1.61

The inherited results are not included, as attention is focused on the overheads needed to collect these results.

7.3.2 Summary of the cost-centre-stack table

At the end of the program execution 45 cost-centre stacks are produced. The largest of these stacks is 7 cost centres deep. By extending the output produced by the cost-centre-stack profiler, the index tables of these cost-centre stacks can be analysed.

The index tables of these stacks vary from size 0 to size 12. 84.4% of these index tables are of size 0 and 1, a further 6.6% of the index tables are of size 2, 3 or 4. 4.4% of the index tables are of size 6 and there is one index table of size 11 and one of size 12.

Considering the size of the program which is being profiled, the cost-centre-stack table is small; there are a number of observations which can be made with regard to this:

- The physical size of the program (measured by the number of lines) is large;
- However, the call-graph structure of the program is relatively simple. Most of the functions in the program call a small number of other functions - this can be seen from the program's index table;

- The overheads of the program are therefore small as most of the book-keeping needed for the cost-centre-stack profiler is straightforward. Once a stack is created it will be memoised and reused. While the program is executing the C code of the program, the current cost-centre stack will remain static.

These observations lead to interesting initial conclusions - the overheads of the cost-centre-stack profiler are going to be based (partly) on the structure and not simply the size of the program. In this particular case study the overheads are very favorable even though the program is large. The analysis of the overheads in terms of the structure of the program is continued in section 7.6.

7.3.3 Compilation overheads

Heap space needed for compilation:

There is no detectable difference in the heap needed during compilation for the cost-centre profiler and the cost-centre-stack profiler. The makefiles used in each case are the same.

Time needed to compile the program:

A typical example of the time needed to compile the modules in the supporting directories, `utils`, `controls`, and `StaticNet`, and the total time to compile the `ntest` program (including the times for the supporting modules) is:

With cost-centre profiling = `utils` (2136.3 user 251.9 sys) +
`controls` (783.9 user 31.3 sys) +
`StaticNet` (419.4 user 44.7 sys)
Total = 3339.6 user 327.9 sys

These times are recorded using the unix time command; adding the user and system time gives a total compilation time of 3667.5 seconds.

With cost-centre-stack profiling = `utils` (2672.4 user 272.5 sys) +
`controls` (647.3 user 150.6 sys) +
`staticNet` (501.9 user 47.4 sys)
Total = 3821.6 user 470.5 sys

Adding user and system time gives a compilation time of 4292.1 seconds.

This makes the compilation time overheads of using the cost-centre-stack profiler 17%. After 10 compilations of the system using the cost-centre profiler and the cost-centre-stack profiler, the overheads average out as 17.03%. This added overhead is considered to be quite acceptable.

7.3.4 Executable overheads

Size of the two executable files:

To analyse the size of the executable files, it is necessary to consider the size of the archive files in the supporting utility directories. Compilation of the `utils` and the `controls` directory produces two such archive files, `utils.a` and `controls.a`.

Compiling the `ntest` program with the cost-centre-stack profiler produces the following files with the respective sizes (in brackets) recorded in bytes:

With cost-centre profiling = `utils.a` (2924936) and `controls.a` (735436)

`ntest` = 3260416 bytes

With cost-centre-stack profiling = `utils.a` (3629074) and `controls.a` (817758)

`ntest` = 3448832 bytes

The difference between the two `ntest` executable files is therefore 188416 bytes, which makes the executable size overheads in this case 5.77%. For a 3.5Mb executable, an extra 1.9Kb seems an acceptable overhead.

Run-time speed and memory usage:

Speed:

On average, execution time using the cost-centre profiler takes a total of 4.5 seconds; with the cost-centre-stack profiler the average execution time is the same. There are no differences in the speed at which the program executes using the two profiling schemes.

Heap usage:

The heap usage for the cost-centre profiler and the cost-centre-stack profiler is also the same. Each will run in 4Mb of heap space.

The results from a large functional program are extremely encouraging. The study is strengthened with the results of profiling the LOLITA system.

7.4 LOLITA

The LOLITA system is one of the largest test cases available for profiling. The version of LOLITA which is profiled to gather the results for this thesis contains 39094 lines of Haskell code and 10177 lines of C code. In addition, the system contains 6.79Mb of data. The version of LOLITA is written in Haskell 1.2, it is compiled with GHC 0.22 and it is a version of GHC 0.22 on which the cost-centre-stack profiler is implemented.

7.4.1 LOLITA results

First example

The LOLITA system is interactive and offers a number of operations to its user. These operations will invoke different parts of the system and consequently will produce different results during profiling. Before any of these operations can be performed the system must load its semantic-net data, a process previously doc-

umented as a computationally expensive operation; see chapter 4. At the end of an execution, the LOLITA system saves the semantic-network data structure. As these two operations are required each time the LOLITA system is run, they provide a suitable test case for the cost-centre-stack profiler. It is noted that analysis of the cost-centre profiler began with these same tests.

The cost-centre stacks are a good deal more complicated than for the previous examples, yet even with this set of results, it is still possible to quickly identify the stacks associated with a high cost[8].

The programmer's attention is drawn to the cost-centre stack with the highest costs:

```
<StaticNet_load_Binary,StaticNet_sNetLoad,StaticNet_sInitialiseData,
 Total_loadData,Okf_mapOKF,IMain_go,> with  157 TICKs
```

At the head of this stack is the cost centre `StaticNet_load_Binary`. This cost-centre stack alone accounts for 24.3% of the total execution time. These large costs are due to the loading of the LOLITA semantic-net data structure. The sequence of cost centres to this particular point in the program is clearly shown.

The investigation of the cost-centre-stack profile is facilitated by the use of the graph-tool and post-processor. There are a considerable number of cost-centre stacks to analyse and perhaps only the most diligent programmer is prepared to look through the cost-centre-stack output to find the expensive stacks. The output is loaded into the graph-tool. A screen dump of the graph-tool results is shown in Figure 7.14.

The resulting graph contains 66 different nodes and 83 links between these nodes. For this reason the graph is large and can not be seen in a single window display. The screen dump shows the top twenty-one nodes in the graph; the virtual display window, in the bottom right-hand corner, has a small box around the part of the graph which is currently being displayed. This accounts for under a quarter of the total graph size.

[8]The long list of cost-centre stacks is not included in this thesis.

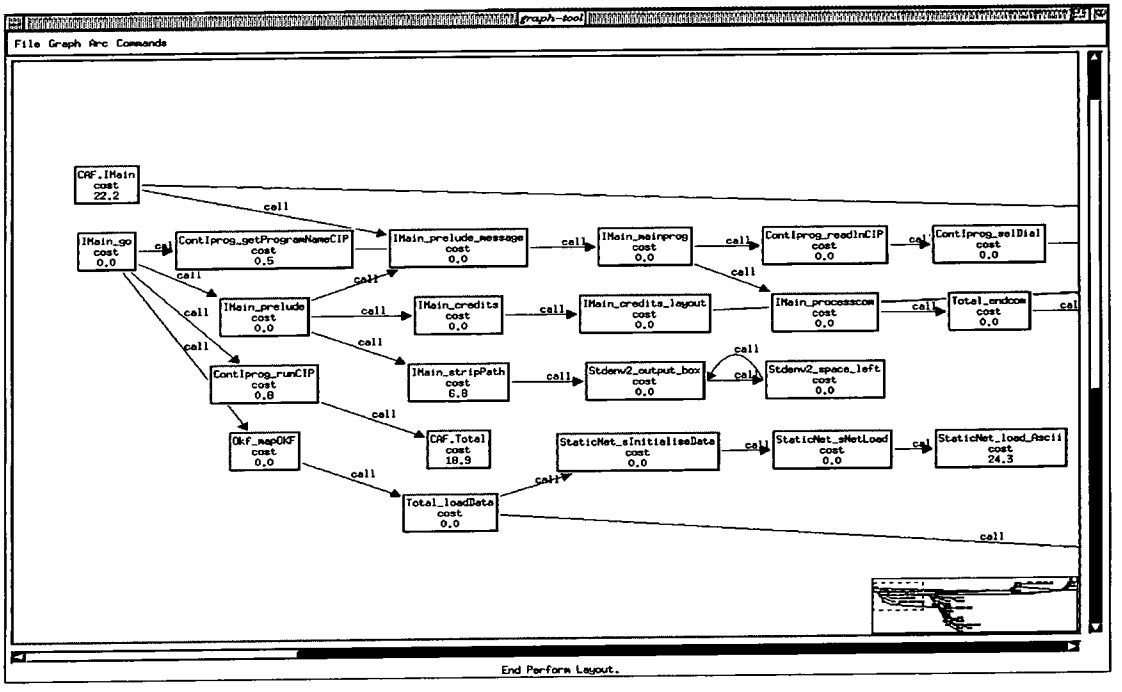


Figure 7.14: Graph-tool display of LOLITA's cost-centre-stack profile results.

The programmer is able to move through the graph display using the scroll-bars provided. In this way the results graph can be seen section by section. This particular display allows the programmer to see the sequence of cost centres which account for the loading of the semantic net, and 24.3% of the total time as displayed at that node.

Viewing the results in this way may look complicated. To improve the situation, the graph-tool is also able to produce two orientations of the complete profiling results; reduced copies of these views can be dumped to a printer. Viewing the complete graph of the program gives the programmer further opportunity to explore the results. With just a few hours experience the programmer will find that the profiling results can be read with relative ease and the expensive parts of the program can be rapidly identified with remarkable accuracy.

The programmer may find that the inheritance and graph-tool functions are an easier way of managing the profiling results. Consider the following three examples of post-processing:

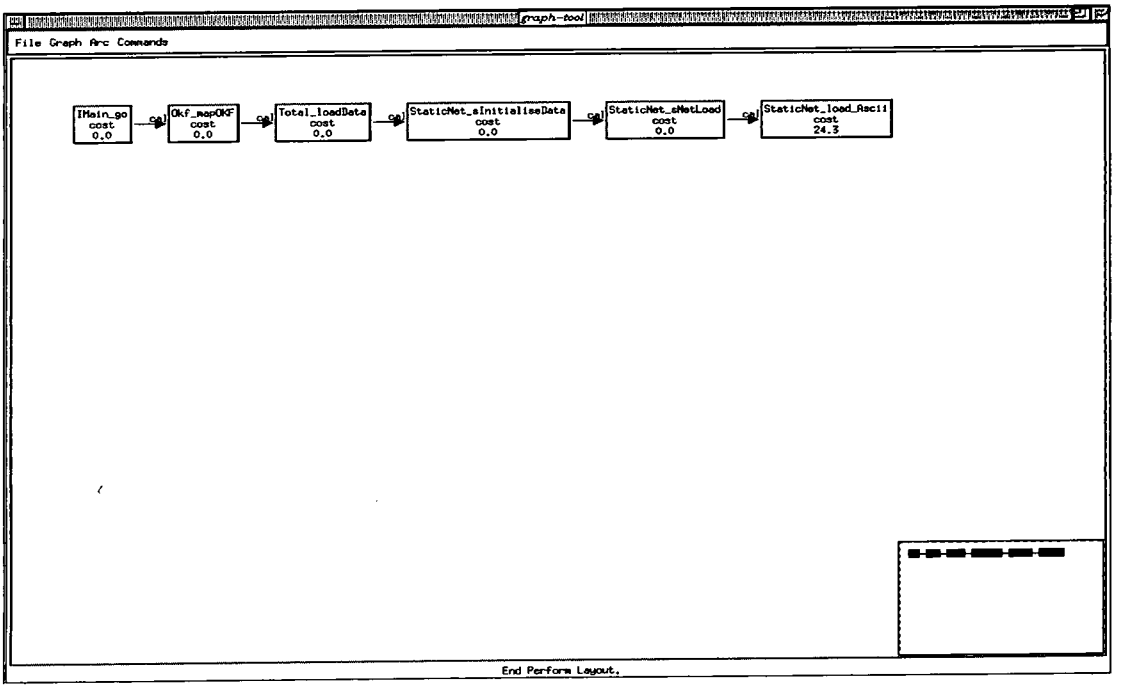


Figure 7.15: Graph-tool view of the most expensive cost-centre stack.

- Using the post-processing tool the programmer can select the cost-centre stack with the highest associated costs. This is effectively the most computationally expensive part of the code, as indicated by the profiler. This information is straightforward to interpret. Figure 7.15 shows this function applied to the LOLITA results; there can be no question as to where in the program the largest amount of time was spent. Other post-processing functions could conceivably be developed which would add arms to the graph one by one, corresponding to the descending order of costs from the cost-centre stack profile.

- A simplified view of the proposed function above is to display only the parts of the graph which have non-zero costs. A reduction of the graph of the LOLITA system, to only those cost-centre stacks with associated costs, is shown in Figure 7.16. The graph has been reduced by more than a half; it now contains 31 cost centres and 30 links between nodes. It is clearly easier to read the results in this form as they appear on a single screen. In general, a programmer will only be interested in a profile of the program which shows the actual program costs.

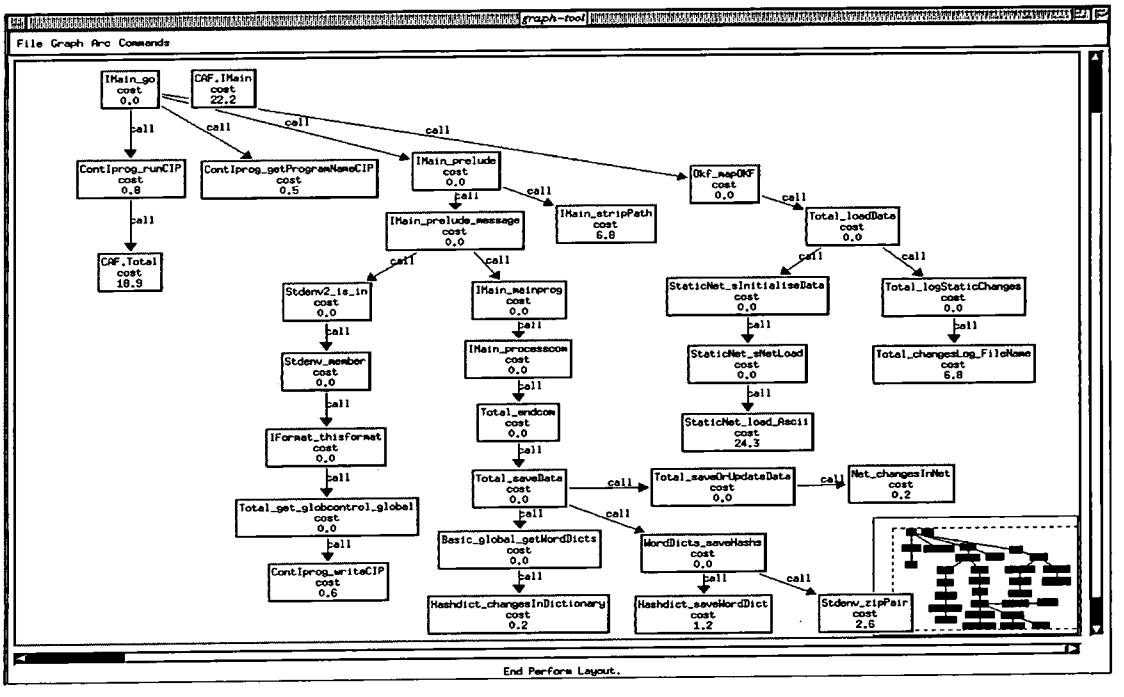


Figure 7.16: Graph-tool view of all stacks with non-zero costs.

- Post-processing also allows the programmer to select particular functions in which he is interested in profiling; this was one of the earlier assumptions made in the thesis, see chapter 6. This facility is demonstrated on the LOLITA cost-centre-stack profile by selecting the following four cost centres: `Total_loadData`, `IMain_go`, `IMain_prelude` and `Total_saveData`. It may be useful for the programmer to gather profiling costs in terms of these four functions, as it allows the developers to see how much time is spent loading and saving the semantic net; how much time is spent in the prelude function of the `IMain` module and how much time is spent in the main function `go`. Those functions which are not selected have their costs subsumed by those functions which are selected; see Figure 7.17. The costs do not account for 100% of the overall costs due to some of the CAF costs; this is discussed.

These post-processing facilities allow the programmer to explore the profiling costs after program execution. In this example, the programmer is able to see that loading the semantic-net data accounts for 31.1% of the total execution time. The LOLITA prelude accounts for 7.4% of the program costs; this involves formatting and printing the credit information and information regarding the authors of the

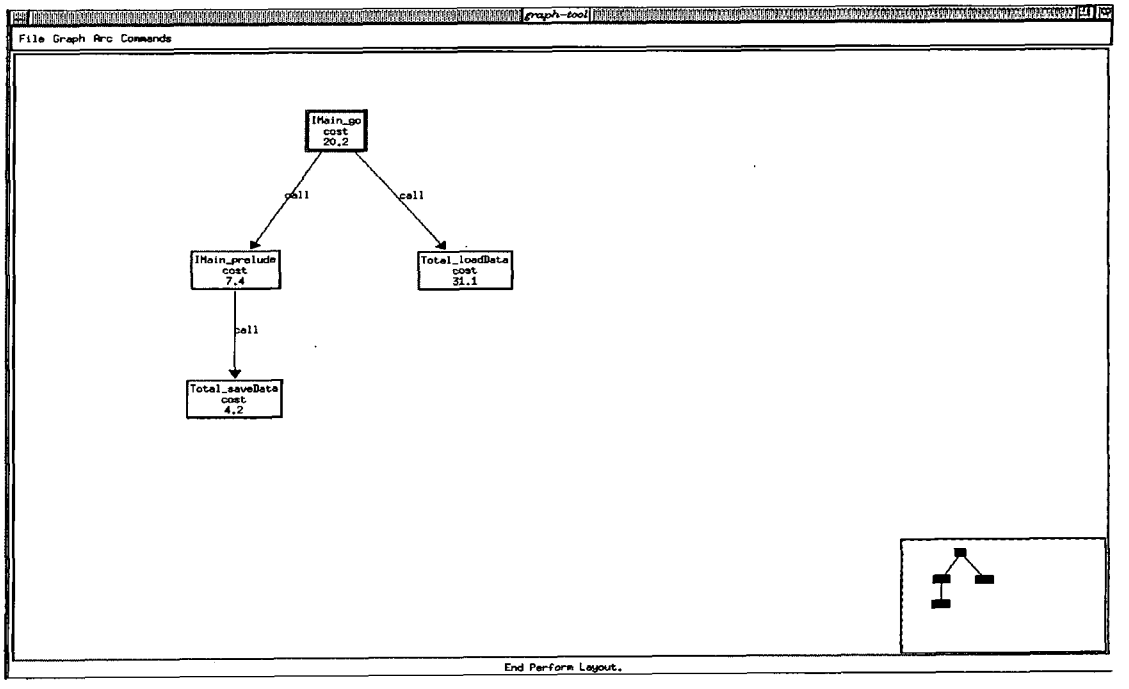


Figure 7.17: Graph-tool view with selected cost centres.

system. At least 4.2% of the program's execution time is due to saving the semantic-net structure. Some of the saving costs (18.9%) have been attributed to a CAF in the `Total` module; this can be seen in Figure 7.16, but not in Figure 7.17. It is expected that later versions of the cost-centre-stack profiler will improve this situation; later versions of the GHC cost-centre profile are more explicit regarding the cost information for CAFs[9]. A further 20.2% of the program costs are subsumed to the cost centre `go`; this corresponds to the `main` function of LOLITA.

These results show a clear mechanism by which a programmer can collect and view profiling results. The post-processing facilities are clearly useful in such a large example and the way in which they are able to filter large collections of results gives the cost-centre-stack profiling scheme potential.

It is proposed that, with continued testing and feedback, a suite of post-processing functions could be developed to make the interpretation of thet cost-centre-stack results easier. As most of these post-processing functions are written in C, it is feasible for a programmer to develop his own post-processing functions.

[9]Now that the prototype has been tested and has provided satisfactory results, it is proposed that the cost-centre-stack profiler is implemented on GHC 2.01.

Input	Comp time	Exec size	Runtime	Stacks	Depth	Push
(*i*)	50123.7 (10.2%)	41279488 (7%)	110.9 (70.6%)	1807	112	278081
(*ii*)	same	same	55.4 (130.8%)	1766	94	12321

Table 7.1: LOLITA results for template analysis

Second example

The second example considers a LOLITA function which makes use of a far greater percentage of the total code. Template analysis takes as its input a passage of text. This is then parsed and semantically analysed to produce a network of semantic nodes from which information can be scanned to match a collection of templates.

Two sets of input data were tested. The first was a passage of text taken from the Telegraph newspaper concerning one of the IRA terrorist incidents (*i*); it contained 74 words. The second piece of data was the sentence "The cat sat on the mat" (*ii*).

The first set of data clearly required more processing than the second, though using these two sets of data allows the comparison of the profiling overheads required for each. Most of the time taken in (*ii*) is due to the loading of the semantic network.

Table 7.1 shows: the input data used; the time taken to compile the LOLITA system using the cost-centre-stack profiler (the difference between this and the cost-centre profiler is shown in brackets); the size of the executable file produced by the cost-centre-stack profiler (the difference is again shown in brackets); the template-analysis run-time (again the difference is in brackets); the number of cost-centre stacks produced as output; the depth of the largest cost-centre stack and the number of Push operations performed during program execution.

As expected the larger input causes more computation and as a consequence of this the mechanics of cost-centre profiling are considerably more detailed - this is shown by the number of Push operations performed during each execution. The depth of the largest cost-centre stacks are similar for both sets of input - 112 and 94

cost centres respectively. Since large stacks are constructed for the small input as well as the large, many of the 265760 Push operations (the difference between the two tests) will be performed on memoised stacks - this accounts for the reduction in overheads when considering a larger input.

7.4.2 Compilation overheads

The compilation overheads are shown in the tables below.

LOLITA	*Compilation time (seconds)*		
Directory	*cost-centre profiling*	*cost-stack profiling*	*Overhead (%)*
utils	2320.7	2683.0	15.51
StaticNet	262.4	319.7	21.83
controls	550.1	615.6	11.9
tomita	141.8	149.2	5.04
haskell	40353.1	44496.3	10.27
linking	1846.2	1867.3	1.14
Total time	45481.7	50123.7	10.20

LOLITA	*Archive size (bytes)*		
Directory	*cost-centre profiling*	*cost-stack profiling*	*Overhead (%)*
utils	2927666	3629676	23.97
StaticNet	585456	666316	13.81
controls	750574	847792	12.95
tomita	154630	164200	6.18
haskell (lolita.exec)	38581248	41279488	6.99

The code for the LOLITA system is defined in four sub-directories. The first is for the utilities, `utils`; the `StaticNet` directory contains the semantic-net utilising

functions; then there is a `controls` directory and a `tomita` directory, the latter of which contains the tomita parser.

The first table shows the compilation time, in seconds, for each of the LOLITA directories, for both the GHC cost-centre profiler and the new GHC cost-centre-stack profiler. The table also shows the linking times for the complete system.

The compilation time overheads range from being negligible to 20%. These results are taken from an average of five compilations. The largest directory of files, LOLITA's `haskell` directory, takes over 11 hours to compile all the modules. The cost-centre-stack profiler introduces an overhead of 10%, which means that the user would need to wait another hour for the complete system to compile. This might seem like a long time, but if it is considered that linking the system takes half an hour and compiling a single module takes on average four and a half minutes, then changing the cost centres twice will result in the cost-centre-stack profiler being more efficient.

Experience shows that in practice the programmer moves cost centres around in his code a number of times even when dealing with a small part of the code. Considering that the cost-centre-stack profiler allows the programmer to explore the costs of the whole program after one compilation, the compilation time overheads for the cost-centre-stack compiler are extremely encouraging.

The second table shows the size of the archive files (.a) for each of the sub-directories and also the size of the LOLITA executable (lolita.exec). The overheads range from 6% to 23% with a total overhead of 7%. A total overhead of 7% seems very reasonable, although this does account for 2.7Mb of extra information. In general the overheads for the cost-centre profiler are high and any improvements to this profiler would have obvious benefits to the size of the executable produced using the cost-centre-stack profiler.

7.4.3 Executable overheads

The execution overheads vary according to the LOLITA functions utilised and the input supplied.

The execution time overheads for the first example over an average of 15 executions are in the order of 13.2%; this only accounts for an extra 2 seconds of execution time. Such overheads are acceptable and show that using the cost-centre-stack profiler will have obvious time benefits to the programmer.

The execution time overheads for the second example are higher and at worst are in the order of 130.8%. Even this however only accounts for a further half a minute of execution time. Using the cost-centre-stack profiler in the detailed analysis of this example will also save the programmer a considerable amount of time.

7.5 `nofib` benchmark

The previous results are further supported by the results of testing the cost-centre-stack profiler on the `nofib` benchmark, [Partain, 1992] [Jarvis and Morgan, 1996a, 1996b]. Partain describes the testing of compilers for lazy functional programs as a 'near-scandalous subject', referring to the ambitious claims made by some implementors based on the results of small test programs, `8queens` and `fib 20` for example. In response to this, Partain gathered some serious benchmark programs which he collectively called the `nofib` suite.

The `nofib` suite specifically consists of:

- Source code for real Haskell programs which can be compiled and run;
- Sample inputs (workloads) to feed to the compiled programs, along with the expected outputs;
- Specific rules for compiling and running the benchmark programs and reporting the results; and

- Sample scripts showing how the results should be reported.

Those programs included in the `nofib` suite are divided into three subsets, Real, Imaginary, and Spectral (between Real and Imaginary).

The Real subset is described as being the most important as it contains programs which are written by someone trying to get a job done, not by someone trying to make a pedagogical or stylistic point. They perform a useful task and are not implausibly small (or large). The run-times and space for the compiled programs are selected so that they represent a reasonable amount of work and the programs are written by a diverse set of people based at different sites, with varying functional programming skills and styles. The programs also span many different application areas and have varying ages. Nine of the programs from the Real subset, whose benchmark results are displayed in this thesis, can be seen in Table 7.2.

Programs in the Spectral subset do not quite meet the criteria of the Real subset. Many of these programs fall into Hennessy and Patterson's category of kernel benchmarks [Hennessy and Patterson, 1990]; they are 'small key pieces from real programs'. Five of these programs, whose results are also included in the benchmark tests, can be seen in Table 7.2.

The Imaginary subset contains programs such as `queens` and `fib` etc. These programs are not considered to be as important [Partain, 1992] and are therefore not referred to in any more detail.

The version of the `nofib` suite used in these tests dates from June 1996. All the tests were performed on the same machine[10]. The standard optimiser (`-O`) was used during the compilation and the compile-time flags were set so that all top-level functions were profiled (`-prof -auto-all`). The compiled programs were run with the time profiler (`-pT`) and the stats option (`-s`) so that the heap and time usage could be recorded.

[10]System Model : SPARCclassic, Main Memory : 96 MB, Virtual Memory : 353 MB, CPU Type : 50 MHz microSPARC, ROM Version : 2.12, OS Version : SunOS 4.1.3C.

Program	Description	Origin
Real subset		
`ebnf2ps`	Context free grammar translator	Peter Thiemann (Tubingen)
`gamteb`	Monte Carlo photon transport	Pat Fasel (Los Alamos)
`gg`	Graphs from GRIP statistics	Iain Checkland (York)
`maillist`	Mailing-list generator	Paul Hudak (Yale)
`mkhprog`	Haskell program skeletons	Nick North (NPL)
`parser`	Partial Haskell parser	Julian Seward (Manchester)
`pic`	Particle in cell	Pat Fasel (Los Alamos)
`prolog`	"mini-Prolog" interpreter	Mark Jones (Oxford)
`reptile`	Escher tiling program	Sandra Foubister (York)
Spectral subset		
`ansi`	Direct cursor input/output	Will Partain (Glasgow)
`banner`	A simple banner program	Mark Jones (Oxford)
`eliza`	The classic pseudo-psychoanalyst	Mark Jones (Oxford)
`minimax`	Tic-tac-toe	Iain Checkland (York)
`primetest`	Primality testing	David Lester (Manchester)

Table 7.2: `nofib` benchmarks: 'Real' and 'Spectral' programs

In the majority of cases the supplied input was used during program execution, although some of the input data was extended to increase the run-time of the programs. The only other changes made to the programs were for debugging purposes (incorrect Makefiles etc.). Not all of the programs included in the suite compiled correctly; some required a more up-to-date version of the compiler (0.24+) and some of the programs had files missing. All of the programs which compiled and ran correctly were included; that is to say, this data was not selected on the basis that it produced favourable results.

For each program tested, the results of compiling and running the program under the cost-centre-stack compiler have been recorded. The difference in the overheads of the cost-centre-stack compiler and a standard version of GHC 0.22 (using the `-prof -auto-all` compiler flags) is shown in brackets. Statistics recorded include compile time, run-time, total memory consumption, the number of cost centres in the program and the number of Push operations performed by the cost-centre-stack profiler. These results can be seen in Table 7.3.

Program	Comp. time sec. (diff.)	Memory usage (diff.)	Run-time sec. (diff.)	Cost centres	#Push
Real subset					
`ebnf2ps`	2518.5 (13.5%)	9,804 kb (61.3%)	3.88 (84.8%)	225	34335
`gamteb`	816.0 (13.2%)	30,712 kb (414.8%)	216.3 (100:1%)	57	356142
`gg`	1097.7 (11.1%)	19,808 kb (229.7%)	14.4 (73.2%)	133	76272
`maillist`	106.8 (3.8%)	4,916 kb (5.4%)	20.1 (6.0%)	10	5367
`mkhprog`	476.3 (13.9%)	5,136 (2.8%)	0.4 (42.8%)	30	134
`parser`	1331.9 (7.7%)	25,592 kb (356.0%)	79.5 (384.7%)	78	1375957
`pic`	475.3 (6.5%)	5,208 kb (19.5%)	11.9 (6.25%)	26	6885
`prolog`	442.0 (12.6%)	12,072 kb (136.5%)	4.3 (358.3%)	64	40847
`reptile`	1116.2 (9.9%)	11,104 kb (101.5%)	7.3 (38.0%)	253	30051
Spectral subset					
`ansi`	141.2 (5.7%)	4,868 kb (0.2%)	0.7 (133.3%)	26	113
`banner`	265.6 (0.2%)	5,096 kb (3.2%)	0.6 (10.9%)	9	3359
`eliza`	284.2 (7.12%)	9,864 kb (98.4%)	2.6 (144.8%)	17	24696
`minimax`	379.0 (6.6%)	7,684 kb (55.7%)	5.0 (152.3%)	40	114384
`primetest`	216.6 (6.0%)	9,428 kb (89.0%)	153.3 (2.4%)	21	24062

Table 7.3: `nofib` benchmark results

7.5.1 nofib results

Compile time - between 3.8% and 13.9% overheads. This is due to the time needed to produce the larger executable file. These overheads are considered reasonable. The size of the executable files was expected to be slightly larger because of the changes made to the compiler optimiser and to the run-time system.

Memory usage - previous examples have shown that programs compiled with the cost-centre stacks use the same amount of heap as those compiled with cost centres. The overall physical memory usage is greater however; memory overheads range from 0.2% to 414.8% depending on the structure of the program. This can cause a large amount of extra memory to be needed during execution[11]. However, these overheads have never prevented a program from being executed on a normal Sparc workstation.

Run-time difference - this is where the most overheads are anticipated, as most of the profiler changes are to the run-time system. These range from 2.4% to 384.7%. These overheads are also dependent on the structure of the program (see section 7.6).

Even when the run-time overheads are 384% (`parser`), this only means an extra 59 seconds of execution time, which accounts for just 4.4% of a single compilation of that program.

The relation between, the number of cost centres and Push operations, and the overheads is discussed in the next section.

These results show that the cost-centre-stack profiler should be used if the cost centres are going to be moved once or more in the analysis of a program. This allows a substantial amount of time to be saved.

[11] An extra 23Mb in the worst case.

7.6 Complexity analysis

How can the difference in run-time overheads found in the testing of the cost-centre-stack profiler be explained? The run-time overheads are dependent on the structure and style of the program. For example,

- the complexity of the cost-centre-stack table will increase with the number of arcs in the call-graph. The greater the functional dependency, the greater the overheads involved in creating the cost-centre stacks;
- the more cost centres there are per unit of code, the greater the overhead of managing the cost-centre stacks. The number of top-level functions may be increased by the programmer's style, for example, if the programmer is not keen on local function definitions.

It is important to note that it is not simply the size of a program which increases the overheads. This is shown by the LOLITA results, for example, where the overheads are lower than the overheads of programs hundreds of times smaller. It is the possibilities allowed in the call-graph, which are fulfilled at run-time, that increase the overheads.

Of course this analysis (as in the `nofib` results above) is based on the assumption that all top-level functions are profiled (`-auto-all`). This need not always be the case. The cost-centre-stack profiling method is equally valid with less cost centres in the code, in fact the programmer might find high-level information, gained from a profile containing less cost centres, more informative at the early stages of profiling. It would also be the case that the use of less cost centres would reduce the overheads of the cost-centre-stack profiler.

7.6.1 Worst-case analysis

A worst-case analysis of the complexity of the algorithm can be calculated. Assume that there are n cost centres in the program and that the worst-case scenario is

modelled in which every function can (and does) call every other function in every possible way (i.e. with every possible combination of enclosing cost-centre stack).

- *Space complexity*: The number of cost-centre stacks becomes the number of ways of ordering n identifiers, which is $n!$. Each stack must use a constant amount of space (due to stack compression and the fact that all the other stacks will be present). Each stack will also contain a cost-centre-stack index table containing $O(n)$ entries. Hence,

  Space complexity = $O(n!)$;

- *Time complexity (Building stacks)*: This figure will be at least $O(n!)$ because the system would need to build all of the stacks above. The time complexity also involves the addition of the complexity of the Push operations. If it is considered that the number of Push operations performed is c then:

  If the stack already exists the complexity is $O(c*n)$, n being the size of the index tables;

  If the stack does not exist ($O(n)$ to determine this) then the algorithm searches down the stack looking to see if that function has already been called. In the worst case it would have to search to the end of the stack $O(n)$. It would then have to rebuild the stack, $O(n)$ at worst. This makes a total complexity in $O(c*n)$.

This again is working on a worst-case scenario - with more detailed analysis it could be calculated that not every stack would need to be searched completely, just as every index table would not need to be searched completely.

This analysis opens the way for many optimisations of the algorithm. For example the index tables could be ordered in such a way as to minimise the search needed. Straight recursion could be modelled by storing a functions index pointer to its own stack at the first reference in the index table; reducing the time needed to Push recursive calls to a constant complexity $O(1)$.

It is assumed that the number of Push operations is c; this figure is calculated:

- *Time complexity (Push operations)*:

  This must be at least *O(n!)* to allow for every possible combination of cost-centre stacks. This figure is only interesting however if it is proportional to the execution time overhead, that is in terms of the original computation.

  As a proportion of the execution time overheads for a normal execution of the program, *c* is *(c+c*3n)/c*, thus making the overhead a factor, *1+3n*. This gives a linear relation between the overheads and the number of cost centres (in the very worst case).

Of course, no 'real' program would be this bad. Most programs will have this figure limited by the possibilities allowed in the call-graph. However, this does give an upper limit to the algorithm so it is a useful calculation to have to hand.

7.6.2 Program structure

An average-case analysis of the algorithm would not eliminate the largest figure in the order of complexity, that is the factorial figure which corresponds to the number of stacks produced in the execution of the program.

However, if the shape of the program is different then this factorial figure is already much smaller. For example if the call-graph is a tree, where each function calls only one other function, then the number of stacks becomes n and the space complexity is *O(n)*.

The results in table 7.3 show that there is not a direct correspondence between the number of cost centres, Push operations and the overheads involved in using the cost-centre-stack profiler as opposed to the cost-centre profiler. These overheads, as discussed, are partly based on the structure of each individual program, detailed analysis of which is beyond the scope of this thesis.

7.7 Heap profiling and serial time profiling

The results presented in this chapter are based on an implementation of the cost-centre-stack profiler which records time profiling information. An extension to the cost-centre stacks, to allow the additional storage of heap profiling information, is straightforward; this too is based on the underlying cost-centre profiler.

The extra overheads which heap profiling will impose are 4 bytes per cost-centre stack, this is represented by an integer in which the number of memory allocation samples is stored. The run-time system will also be required to execute one extra instruction per profile sample; this extra instruction increments the number of memory allocation samples for the cost-centre stack. These extra overheads are thought to be minimal.

The data structure of the cost-centre stack

```
struct CostCentreStack {
                costCentre CostCentre;
                int time_ticks;
                CostStackTable PreviousStack;
                IndexTable CostCentreStackIndexTable;
};
```

is extended to include this extra field.

```
struct CostCentreStack {
                costCentre CostCentre;
                int time_ticks;
                int mem_allocs;  (* Extra heap profiling info *)
                CostStackTable PreviousStack;
                IndexTable CostCentreStackIndexTable;
};
```

Heap profiling results may similarly be produced in hp2ps format, from which they can be displayed as a postscript graph; they may also be displayed in the graph-tool.

From our experience, serial time profiling has not proved to be as useful as time or heap profiling. There may be benefits in extending the cost-centre-stack profiler

so that serial time profiling is implemented, however without any explicit indication from applications programmers that this is useful, it is not proposed that this is explored in any further detail.

7.8 Chapter summary

A prototype of the cost-centre-stack profiler, implemented on GHC 0.22, has produced encouraging results for programs containing 15 to 50,000 lines of code. A number of examples were presented in this chapter which included the Clausify benchmark program, a subset of the LOLITA system called `ntest` and the LOLITA system itself. The results of profiling the `nofib` benchmark suite were also presented.

A small example was introduced at the beginning of the chapter which demonstrated how the cost-centre-stack profiler recorded profiling results in such a way as to make accurate inheritance possible. Even in a simple 15 line program the difference in results between the cost-centre-stack profiler and the cost-centre profiler was considerable.

As the size of the program grew, so the results became more complicated to analyse. For this reason a number of post-processing functions were developed which enabled the programmer to filter the cost-stack results; these included functions which would remove cost-centre stacks with zero costs and a function which would display the cost-centre stack with the highest associated costs. The LOLITA case studies showed that the post-processor was a useful tool in the analysis of profiling results from a large functional program.

The overheads of using the cost-centre-stack profiler were presented. The results showed that the cost-centre-stack profiling overheads were not based on the size of the program but on the program structure. Never were the overheads so high that it was not possible, or useful, to profile a program with the cost-centre-stack profiler. Worst- and average-case analysis of the overheads were discussed.

Further studies will be needed to see if programmers use cost-centre-stack profiling in preference to cost-centre profiling. The case studies have shown that if a programmer is likely to move cost centres in his code to produce successive cost-centre profiles, then using the cost-centre-stack profiler will have considerable benefits.

Chapter 8

Further Research

8.1 Introduction

The cost-centre-stack profiler is based on a well-formed theory; cost-centre stacks are sequences of labels in a program which may correspond to top-level function definitions or individual unnamed expressions. During a program's execution these cost-centre stacks are built in such a way as to ensure that recursive programs are represented by stacks which include only one occurrence of each cost centre. The initial prototyping of the cost-centre-stack profiler has shown encouraging results; in particular the overheads of the scheme are promising.

There is a need for the careful analysis of the new cost-centre-stack profiler in a series of detailed case studies. The cost-centre-stack profiler has already been tested on the LOLITA system; it is proposed that this study is continued and the feedback from developers used in the tuning of the profiling tool. It is also proposed that the profiler is distributed with the Glasgow Haskell Compiler. This will allow testing of the new profiler by a wide variety of programmers.

The development of this method of profiling has led to two important advances. Firstly, profilers can be constructed which obtain a much wider range of data from a single profile of a system. Further research includes the development of other profilers based on this method, see the BSP tools project described in section 8.4.

Secondly, cost-centre stacks can be used as a basis for the development of more tools needed in the analysis of lazy functional programs. In particular, further research includes the development of tracing and debugging tools, two areas which have seen a considerable amount of research interest. These areas are discussed in sections 8.2 and 8.3.

8.2 Debugging

Chapter three (section 3.5.1) discussed the development of debugging tools which were used in the engineering of the LOLITA system. One of these tools in particular was the Distinguished Path Debugging Tool developed by Hazan and implemented on the Gofer and Miranda interpreters. Since the LOLITA system requires compiler support it is desirable that the Distinguished Path Debugging Tool is ported to either GHC or HBC. The following section re-introduces the Distinguished Path Debugging Tool and gives an indication of how the cost-centre-stack profiler can be used as a basis for an implementation of the tool on the Glasgow Haskell Compiler.

8.2.1 Distinguished Path Debugging Tool

A class of run-time error which was found to be occurring frequently in the LOLITA system was the *exception error* type. An exception error is one which results in termination of the program and the printing of an error message. Examples of this type of error in Haskell are

```
Fail: head{PreludeList}: head []
```

which results from passing the list head function an empty list and

```
Fail: (!!){PreludeList}: index too large
```

which occurs when the list indexing operator is passed a subscript which is outside the bounds of the list. These exception errors give no indication to the programmer

of whereabouts in the program the error occurred. This is a problem, as functions such as head and the list indexing operator are used many times in many different parts of LOLITA. Previously, this problem had been approached by providing a customized version of each function capable of generating an exception error for each module. This new version of the function would report the name of the module when it generated an exception error. However, simply knowing the name of the module in which the exception was generated is not sufficient—the error causing the exception error to be generated may be in a function which called the exception-generating one, or even some way back in a chain of functions each calling the other with the exception-generating function at the very end of the chain.

The *distinguished path debugging tool* [Hazan, 1993] allows the display of these chains of functions, termed *distinguished paths.* The path displayed is the route taken through the dependency graph of the functions in the program, with cycles removed by grouping together mutually recursive functions. For recursive calls, the function is only included once in the distinguished path. The tool works by transforming each function to take an extra parameter, a representation of the distinguished path, which is built up from one function call to the next. When an exception error is encountered, the value of this extra parameter is displayed. Unlike the previous method of debugging such errors, which involved altering the source code by hand, the tool works automatically by transforming each source module.

The example shown in chapter 3 was that of the map colouring problem. The dependency graph of the sample program is shown in Figure 8.1. When the expression `go` is evaluated, the function `addcol` is used to produce a new colouring scheme given the map `country_map`. However, evaluation terminates with a head of empty list error. The distinguished path debugging tool causes the following error message to be output[1]:

```
Program error: hd []: hd lookup will_clash clash addc addcol
```

[1]This output is from a Miranda implementation of the tool.

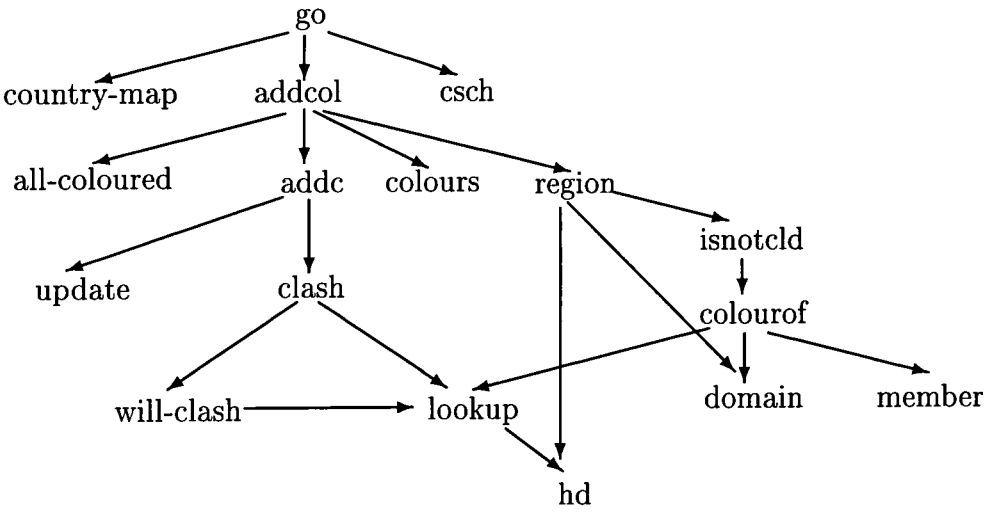


Figure 8.1: The dependency graph of a program to solve the map colouring problem.

This error message shows the distinguished path. The distinguished path may either be searched in a top-down manner, starting from `addcol`, or in a bottom-up manner, starting from `hd`. For the top-down search, each function is searched in turn until a correct function is found; the function containing the error is either the function searched immediately before this one, or another function which is not on the distinguished path but which this function depends on. Alternatively, the bottom-up search proceeds until an incorrect function is found. In the case of the example program, the error is in function `will_clash`, which is on the distinguished path itself.

Experience with this tool has shown it to be very useful in debugging exception errors.

8.2.2 Using cost-centre stacks for debugging

It is proposed that the cost-centre-stack profiler be modified to implement the distinguished path debugging tool on the Glasgow Haskell Compiler. The *distinguished paths* identified by Hazan can be recorded using cost centres, the cost-centre stack being itself a distinguished or unique path through the programs execution.

When a program terminates with an execution-time error, it is possible to

output the current cost-centre stack which is currently in scope. This will have the effect of outputting the sequence of cost centres which were encountered to reach that part of the code where the error occurred.

Recursion is tackled in a different way to the Gofer implementation of the distinguished path debugging tool. The cost-centre-stack profiler is able to provide information on which functions in a mutually recursive function group have been called and which have not; the distinguished path debugging tool is unable to provide this information, it will simply identify the name of the recursion group. The compressed cost-centre stacks mean that the path of cost centres does not include repetitions of the cost-centre name, even when recursive and mutually recursive functions are used.

The modifications to the cost-centre-stack profiler should produce better results to that of the distinguished path debugging tool as the profiler can supply information on terminating as well as non-terminating programs. The profiler also has the added advantage that it can be used with the Glasgow Haskell Compiler and not just with an interpreter-based system.

8.3 Tracing lazy functional computations

Lazy functional programs are noted for the fact that the order of evaluation is determined by the demand for results. This behaviour is hidden from the programmer. Sometimes, however, it is useful to be able to 'see' this activity and watch the order of evaluation of expressions in the execution of a program. In the past the only way to see the sequence of lazy evaluation was to watch the program stack. This was not easy and was difficult to interpret even for the experienced programmer.

Another approach used to trace functional programs has been to provide a visualisation aid to show the order of graph reduction in a program [Foubister and Runciman, 1995].

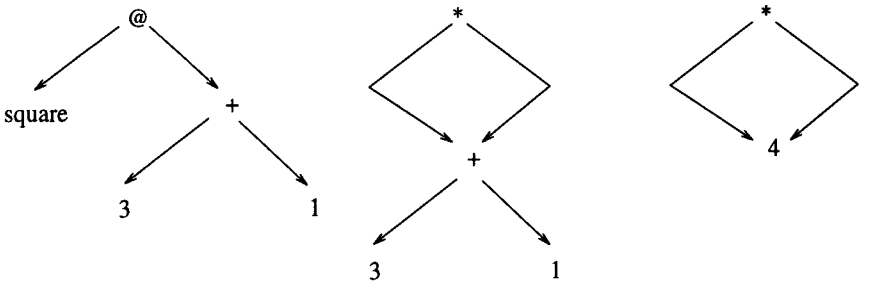


Figure 8.2: The reduction of `square (3 + 1)`.

8.3.1 Visualisation of graph reduction

Foubister and Runciman stated that people are often unable to predict the behaviour of lazy functional programs and, although the order of reduction is deterministic, it is not always intuitive. Their proposed solution makes the details of reduction open to inspection.

In Foubister's thesis it is demonstrated how a visual presentation of graph reduction can be achieved. This allows the presentation of a series of reduction steps such as those shown in Figure 8.2.

Some of the complexity introduced by the crossing of arcs in the reduction graphs is reduced by displaying the results as graph-trees. It is also recognised that the graph-trees may become quite large when more complicated programs are introduced and that the sequence of reduction steps may also create a long series of trees. Solutions to these problems were implemented in the form of filtering the trees, to reduce the number of vertices, and reducing the number of graphs by only showing 'stages of interest'.

Filter functions have been defined by Foubister and as these are simply written in a functional code it is proposed that users can write their own filter functions.

The system is currently written using a subset of the Haskell language called `h` running on an interpreter called `hint`. The scaling-up of the system to full Haskell has yet to be done.

8.3.2 Using cost-centre stacks for tracing

The cost-centre-stack profiler is currently being extended to offer a new way of tracing higher-order computations. The advantage of this is that a full Haskell program can be observed during execution.

The approach to tracing using cost-centre stacks is quite different to that adopted by Foubister for a number of reasons. Firstly, the full Haskell language is used so there is no need for any scaling-up of the language; secondly, it avoids displaying the evaluation of a program in terms of graph reduction, as it is not clear that this will make the evaluation of a program any more intuitive, particularly if an optimising compiler is used; finally, the emphasis is on programming on a large scale and making a tracing tool which is applicable to programs which are potentially thousands of lines long.

With the cost-centre-stack tracing mechanism it will be possible to study the sequence of evaluations by looking at the order of the cost centres in the cost-centre stacks at different points in the programs execution. This gives programmers the chance to watch the lazy evaluation sequences of programs. There are some obvious benefits which this kind of tool would offer to the study of lazy computations:

- it would allow compiler writers to experiment with different evaluation and optimisation methods in their compilers to reduce the size of the stacks;
- functional programmers would in the same way be able to experiment with different programming methods to optimise the way in which their programs were executed;
- students would be able to gain a picture of how a simple program does actually evaluate using a lazy evaluation mechanism. All people interested in lazy evaluation would have a simple way to look at the method in action.

The tracing problem is receiving a good deal of attention from York and Chalmers Universities, they recognise the value of such a tool in compiler research and also in the development of support tools for lazy functional languages.

The cost-centre-stack profiler will provide the necessary mechanism for the implementation of such a tool on the Glasgow Haskell Compiler.

8.4 BSP tools project

The goal of the BSP tools project[2] is to develop tools and libraries which facilitate programming within the Bulk Synchronous Parallel (BSP) model. The BSP model provides a simple, unified framework for the design and programming of all kinds of general purpose parallel systems [McColl, 1995]. The motivation behind this project is the need for a unified framework in which parallel software can be developed on a diverse range of architecture.

Traditionally computer architectures have developed in many different directions and produced many different kinds of memory architectures and multiprocessing systems. Developing software which will run on any general-purpose parallel architecture might seem an increasingly unlikely prospect. However, the BSP project aims to produce a framework for the design and programming of general purpose parallel systems. In addition to this the software produced should be fully portable parallel software which is efficient, scalable and has a predictable performance on a range of parallel architectures.

The first stage of the BSP tools project has been to provide new native implementations of the Oxford BSP library for an SGI Power Challenge and an IBM Sp2. A generic implementation has also been constructed on top of MPI (Message-Passing Interface), a new standard for writing portable message-passing programs.

New libraries have been created for these systems: along with the new implementations of the BSP library, a collection of higher-level libraries have also been developed. In particular,

- a library of monolithic array processing operations similar to those available

[2]An on-going project at the Programming Research Group, Oxford University — ESPRIT Basic Research Project 9072 - GEPPCOM (Foundations of General Purpose Parallel Computing).

in High Performance Fortran (HPF) and the Bird-Meertens formalism. The motivation for such a library is to provide monolithic operations such as filter, scan, and scatter which view a collection of distributed arrays as a single monolithic unit.

- a library that allocates, and maintains references to, dynamic heap-allocated objects on remote processors. Operations are provided for remote dereference, store, and the wholesale pulling of objects from one processor onto another.
- a library that enables the farming of arbitrary tasks between processors.

The next stage of the project is to investigate profiling and debugging tools. It is proposed that the cost-centre-stack profiling method be used as the design for these tools. The BSP compiler uses a number of techniques adopted from the field of lazy, higher-order functional languages. For example many of the message-passing instructions are based on a delayed evaluation mechanism, for this reason it is not always easy to predict where much of the communication in a program will take place. Run-time analysis is possible with a profiler designed for a lazy functional language; the BSP compiler is currently being modified to output process stacks. The post-processor is also considered to be highly desirable by BSP programmers and developers.

Chapter 9

Conclusion

The conclusions to the thesis begin by recalling the criteria for success which were discussed in Chapter 1.

1. The new method of profiling presented in this thesis, provides an opportunity for a reduction in the time needed to profile a large lazy functional programmed system, when the programmer selects and deselects parts of the code for profiling.

2. The new method of profiling extends the profiling results presented by previous profilers in so much as the accurate inheritance of shared program costs can be achieved.

3. The new method of profiling provides these new facilities without imposing an unacceptable overhead on the compilation or execution of the program such that the new method of profiling would offer no benefit to a functional programmer.

The development of the cost-centre-stack profiler was based on the results of a series of case studies implemented over a two-year period. The case studies investigated the profiling of the LOLITA system, a large-scale lazy functional system written in 50,000 lines of Haskell code. This study highlighted a number of prob-

lems with the current profiling tools and in response to these the cost-centre-stack profiler was designed.

The cost-centre-stack profiler collects results which can be post-processed after the execution of the program. Part of the post-processor implements a scheme whereby the programmer can select and deselect cost centres in his code. This enables the results to be viewed at different points in the program without any further compilation or execution of the code.

Experience of using current profiling tools has shown that results are often presented at a low level in the code. Those results which are displayed at a more abstract level in the code often require further investigation to identify exactly where the efficiency problem occurs. The cost-centre-stack profiler will allow results to be displayed at an abstract level in the code and will also allow the re-investigation of costs without any further compilation or execution of the program. This will allow the time spent profiling a program to be reduced. This statement is particularly true as far as large functional programs are concerned, where the compilation and execution overheads might normally be high. This satisfies criterion 1.

The cost-centre-stack profiler provides a method of cost inheritance. Program costs are collected during execution and assigned to cost-centre stacks, an ordered collection of cost centres which show the path of cost centres to that part of the program. These cost-centre stacks provide the basis for a scheme of inheritance which is implemented in the profiling post-processor. The cost-centre stack results can be utilised to provide flat profiles, equivalent to those produced by the cost-centre profiler, and also inheritance profile results. This allows the programmer to investigate the costs of shared functions and leaves him in no doubt as to which parts of the program may have caused an expensive call to a shared function. This satisfies criterion 2.

The cost-centre-stack profiler is designed to minimize the overheads which the collection of this new stack information might impose. A key design issue is that of compressed stacks, which ensure that a cost centre will only appear once in a cost-centre stack. Cost-centre stacks are created using pointers which reduce

the storage overhead needed to implement the cost-centre stacks in the Glasgow Haskell Compiler. Repeated stack push operations are implemented via a high speed look-up table and pointer traversal mechanism. The overheads for small and large programs are encouraging, in so much as a functional programmer would not need to invest much more time in compiling or running his program to gain these extra results. The increased heap usage for the cost-centre-stack profiler is minimal. This satisfies criterion 3.

The theory of cost-centre stacks and inheritance is currently being used in the design of a new profiling tool at the University of Oxford. It is thought that this method of collecting and displaying the results will be useful in the design of profilers for imperative and functional languages in both parallel and serial contexts. The cost-centre-stack profiler also provides the basis for a new method of debugging and tracing non-strict functional programs.

Appendix A

Clausify 5

```
-- CLAUSIFY: Reducing Propositions to Clausal Form
-- Colin Runciman, University of York, 10/90
--
-- An excellent benchmark is: (a = a = a) = (a = a = a) = (a = a = a)
--
-- Optimised version: based on Runciman & Wakeling [1993]
-- Patrick Sansom, University of Glasgow, 2/94

> module Main ( main ) where

-- the main program: reads stdin and writes stdout

> main = readChan stdin exit ( \input ->
>        appendChan stdout (clausify input) exit done)

-- convert lines of propostions input to clausal forms

> clausify input = concat
>                  (interleave (repeat "prop> ")
>                              (map clausifyline (lines input)))

> clausifyline = concat . map disp . clauses . parse

-- the main pipeline from propositional formulae to printed clauses

> clauses =  unicl . split . disin . negin . elim

-- clauses = (scc "unicl" unicl) . (scc "split" split) .
--           (scc "disin" disin) . (scc "negin" negin) .
```

```
--           (scc "elim"  elim)

-- clauses = (\x -> scc "unicl" unicl x) .
--           (\x -> scc "split" split x) .
--           (\x -> scc "disin" disin x) .
--           (\x -> scc "negin" negin x) .
--           (\x -> scc "elim"  elim x)

> data StackFrame = Ast Formula | Lex Char

> data Formula =
>   Sym Char |
>   Not Formula |
>   Dis Formula Formula |
>   Con Formula Formula |
>   Imp Formula Formula |
>   Eqv Formula Formula

-- separate positive and negative literals, eliminating duplicates

> clause p = let
>            clause' (Dis p q)       x   = clause' p (clause' q x)
>            clause' (Sym s)       (c,a) = (insert s c , a)
>            clause' (Not (Sym s)) (c,a) = (c , insert s a)
>            in
>            clause' p ([] , [])

> conjunct p = case p of
>              (Con p q) -> True
>              p         -> False

-- shift disjunction within conjunction

> disin p = case p of
>           (Con p q) -> Con (disin p) (disin q)
>           (Dis p q) -> disin' (disin p) (disin q)
>           p         -> p

-- auxilary definition encoding (disin . Dis)

> disin' p r = case p of
>              (Con p q) -> Con (disin' p r) (disin' q r)
>              p         -> case r of
>                             (Con q r) -> Con (disin' p q)
>                                              (disin' p r)
>                             q         -> Dis p q
```

```
-- format pair of lists of propositional symbols as clausal axiom

> disp p = case p of
>          (l,r) -> interleave l spaces ++ "<="
>                   ++ interleave spaces r ++ "\n"

--eliminate connectives other than not, disjunction and conjunction

> elim f = case f of
>          (Sym s)    -> Sym s
>          (Not p)    -> Not (elim p)
>          (Dis p q)  -> Dis (elim p) (elim q)
>          (Con p q)  -> Con (elim p) (elim q)
>          (Imp p q)  -> Dis (Not (elim p)) (elim q)
>          (Eqv f f') -> Con (elim (Imp f f')) (elim (Imp f' f))


-- remove duplicates and any elements satisfying p

> filterset p s = filterset' [] p s

> filterset' res p l = case l of
>                      []     -> []
>                      (x:xs) -> if (notElem x res) && (p x) then
>                                        x : filterset' (x:res) p xs
>                                else
>                                    filterset' res p xs

-- fewer reductions filterset!

filterset2 _ p l = nub (filter p l)

-- insertion of an item into an ordered list

> insert x l = case l of
>              []     -> [x]
>              (y:ys) -> if x < y then x:(y:ys)
>                        else if x > y then y : insert x ys
>                        else y:ys

> interleave xs ys = case xs of
>                    (x:xs) -> x : interleave ys xs
>                    []     -> []

-- shift negation to innermost positions

> negin f = case f of
```

```
>           (Not (Not p))   -> negin p
>           (Not (Con p q)) -> Dis (negin (Not p)) (negin (Not q))
>           (Not (Dis p q)) -> Con (negin (Not p)) (negin (Not q))
>           (Dis p q)       -> Dis (negin p) (negin q)
>           (Con p q)       -> Con (negin p) (negin q)
>           p               -> p

-- the priorities of symbols during parsing

> opri c = case c of
>           '(' -> 0
>           '=' -> 1
>           '>' -> 2
>           '|' -> 3
>           '&' -> 4
>           '~' -> 5

-- parsing a propositional formula

> parse t = let [Ast f] = parse' t []
>           in f

> parse' cs s = case cs of
>               []      -> redstar s
>               (' ':t) -> parse' t s
>               ('(':t) -> parse' t (Lex '(' : s)
>               (')':t) -> let (x : Lex '(' : s') = redstar s
>                          in  parse' t (x:s')
>               (c:t)   -> if inRange ('a','z') c then
>                             parse' t (Ast (Sym c) : s)
>                          else if spri s > opri c then
>                                        parse' (c:t) (red s)
>                          else parse' t (Lex c : s)

-- reduction of the parse stack

> red l = case l of
>         (Ast p : Lex '=' : Ast q : s) -> Ast (Eqv q p) : s
>         (Ast p : Lex '>' : Ast q : s) -> Ast (Imp q p) : s
>         (Ast p : Lex '|' : Ast q : s) -> Ast (Dis q p) : s
>         (Ast p : Lex '&' : Ast q : s) -> Ast (Con q p) : s
>         (Ast p : Lex '~' : s)         -> Ast (Not p) : s

-- iterative reduction of the parse stack

> redstar = while ((/=) 0 . spri) red
```

```
> spaces = repeat ' '

-- split conjunctive proposition into a list of conjuncts

> split p = let
>           split' (Con p q) a = split' p (split' q a)
>           split' p a = p : a
>           in
>           split' p []

-- priority of the parse stack

> spri s = case s of
>          (Ast x : Lex c : s) -> opri c
>          s -> 0

-- does any symbol appear in both consequent and antecedant

> tautclause p = case p of
>                (c,a) -> -- [x | x <- c, x `elem` a] /= []
>                         any (\x -> x `elem` a) c

-- form unique clausal axioms excluding tautologies

> unicl = filterset (not . tautclause) . map clause

-- functional while loop

> while p f x = if p x then while p f (f x) else x
```

Clausify 6

```
-------------------------
-- Reducing propositions to clausal form.
-- Colin Runciman, University of York.
-- Original version, 18/10/90
-- This is Version 6 (see 'New Dimensions' paper).

> module Main(main) where

-- repeat x = let xs = x:xs in xs
--
-- concat = foldr (++) []
--
-- lines [] = []
-- lines xs = let (l,r) = splitat '\n' xs in l : lines r
--
-- splitat c [] = ([],[])
-- splitat c (a:b) = if a == c then ([],b) else
                              let (x,y) = splitat c b in (a:x,y)

> main :: Dialogue
> main ~(Str user : _) =
>   ReadChan stdin : map (AppendChan stdout) (computer user)
>   where
>   computer = interleave (repeat "prop> ") . map clauses . lines

> data StackFrame = Ast Formula | Lex Char
> type Stack = [StackFrame]

> data Formula =
>   Tru |
>   Sym Char |
>   Not Formula |
>   Dis Formula Formula |
>   Con Formula Formula |
>   Imp Formula Formula |
>   Eqv Formula Formula

> type Clause = (String,String)

-- separate positive and negative literals, eliminating duplicates

> clause :: Formula -> Clause
> clause p = clause' p ([] , [])
```

```
-- NB V6 can use : instead of insert.

> clause' :: Formula -> Clause -> Clause
> clause' (Dis p q)       x   = clause' p (clause' q x)
> clause' (Sym s)       (c,a) = (s:c , a)
> clause' (Not (Sym s)) (c,a) = (c , s:a)

-- the main pipeline from propositional formulae to printed clauses

> clauses :: String -> String
> clauses = concat . map disp . unicl . split .
>                    disin . negin . elim . parse

-- push disjunctions beneath conjunctions

> disin :: Formula -> Formula
> disin (Con p q) = con (disin p) (disin q)
> disin (Dis p q) = disin' (disin p) (disin q)
> disin p = p

> disin' :: Formula -> Formula -> Formula
> disin' (Con p q) r = con (disin' p r) (disin' q r)
> disin' p (Con q r) = con (disin' p q) (disin' p r)
> disin' p q = dis p q

> data Cmp = Same | Oppo | Prec | Foll

-- disjunctions of literals, with inbuilt simplification

> dis :: Formula -> Formula -> Formula
> dis Tru q = Tru
> dis p Tru = Tru
> dis p@(Dis p1 p2) q@(Dis q1 q2) =
>   case cmplit p1 q1 of
>   Same -> dis' p1 (dis p2 q2)
>   Oppo -> Tru
>   Prec -> dis' p1 (dis p2 q)
>   Foll -> dis' q1 (dis p q2)
> dis p@(Dis p1 p2) q =
>   case cmplit p1 q of
>   Same -> p
>   Oppo -> Tru
>   Prec -> dis' p1 (dis p2 q)
>   Foll -> Dis q p
> dis p q@(Dis q1 q2) =
>   case cmplit p q1 of
>   Same -> q
```

```
>   Oppo -> Tru
>   Prec -> Dis p q
>   Foll -> dis' q1 (dis p q2)
> dis p q =
>   case cmplit p q of
>   Same -> p
>   Oppo -> Tru
>   Prec -> Dis p q
>   Foll -> Dis q p

> dis' :: Formula -> Formula -> Formula
> dis' p   Tru = Tru
> dis' p   q   = Dis p q

> cmplit :: Formula -> Formula -> Cmp
> cmplit      (Sym c)       (Sym d)  = cmpchar c d
> cmplit      (Sym c)  (Not (Sym d)) = if c==d then Oppo else
>                                                  cmpchar c d
> cmplit (Not (Sym c))      (Sym d)  = if c==d then Oppo else
>                                                  cmpchar c d
> cmplit (Not (Sym c)) (Not (Sym d)) = cmpchar c d

> cmpchar :: Char -> Char -> Cmp
> cmpchar c d = if c<d then Prec else if d<c then Foll else Same

> con :: Formula -> Formula -> Formula
> con Tru q   = q
> con p   Tru = p
> con p   q   = Con p q

-- format pair of lists of propositional symbols as clausal axiom

> disp :: Clause -> String
> disp (l,r) = interleave l spaces ++ "<=" ++ interleave
>                          spaces r ++ "\n"

-- eliminate connectives other than not, disjunction and conjunction

> elim :: Formula -> Formula
> elim (Not p) = Not (elim p)
> elim (Dis p q) = Dis (elim p) (elim q)
> elim (Con p q) = Con (elim p) (elim q)
> elim (Imp p q) = Dis (Not (elim p)) (elim q)
> elim (Eqv f f') = Con (elim (Imp f f')) (elim (Imp f' f))
> elim p = p

> interleave :: [a] -> [a] -> [a]
```

```
> interleave (x:xs) ys = x : interleave ys xs
> interleave []     _  = []

-- shift negation to innermost positions

> negin :: Formula -> Formula
> negin (Not (Not p)) = negin p
> negin (Not (Con p q)) = Dis (negin (Not p)) (negin (Not q))
> negin (Not (Dis p q)) = Con (negin (Not p)) (negin (Not q))
> negin (Dis p q) = Dis (negin p) (negin q)
> negin (Con p q) = Con (negin p) (negin q)
> negin p = p

-- the priorities of symbols during parsing

> opri :: Char -> Int
> opri '(' = 0
> opri '=' = 1
> opri '>' = 2
> opri '|' = 3
> opri '&' = 4
> opri '~' = 5

-- parsing a propositional formula

> parse :: String -> Formula
> parse t = f where [Ast f] = parse' t []

> parse' :: String -> Stack -> Stack
> parse' [] s = redstar s
> parse' (' ':t) s = parse' t s
> parse' ('(':t) s = parse' t (Lex '(' : s)
> parse' (')':t) s = parse' t (x:s')
>                    where
>                    (x : Lex '(' : s') = redstar s
> parse' (c:t) s = if 'a'<= c && c <= 'z' then
>                     parse' t (Ast (Sym c) : s)
>                  else if spri s > opri c then
>                     parse' (c:t) (red s)
>                  else parse' t (Lex c : s)

-- reduction of the parse stack

> red :: Stack -> Stack
> red (Ast p : Lex '=' : Ast q : s) = Ast (Eqv q p) : s
> red (Ast p : Lex '>' : Ast q : s) = Ast (Imp q p) : s
> red (Ast p : Lex '|' : Ast q : s) = Ast (Dis q p) : s
```

```
> red (Ast p : Lex '&' : Ast q : s) = Ast (Con q p) : s
> red (Ast p : Lex '~' : s) = Ast (Not p) : s

-- iterative reduction of the parse stack

> redstar :: Stack -> Stack
> redstar = while ((/=) 0 . spri) red

> spaces :: String
> spaces = repeat ' '

-- split CNF formula into a list of conjuncts

> split :: Formula -> [Formula]
> split (Con p q) = split p ++ split q
> split Tru = []
> split p = [p]

-- priority of the parse stack

> spri :: Stack -> Int
> spri (Ast x : Lex c : s) = opri c
> spri s = 0

-- form unique clausal axioms excluding tautologies

> unicl :: [Formula] -> [Clause]
> unicl = mkset . map clause

> mkset :: Eq a => [a] -> [a]
> mkset = mkset' []
> mkset' s [] = []
> mkset' s (x:xs) =
>   if x `notElem` s then x : mkset' (x:s) xs else mkset' s xs

> while :: (a->Bool) -> (a->a) -> a -> a
> while p f x = if p x then while p f (f x) else x
```

Appendix B

Cost-centre-stack code

The data structures used to represent the cost-centre-stack table are shown in C. This should allow any recreation of the code. The functions with which the cost-centre-stack operations are defined are described in pseudo-code. This should allow the reader to understand the functionality; a conversion to C code is then simply a matter of syntax.

```
/* ============================================================ */

/* ------------------------------ DATA STRUCTURES ------------- */

/* The two structures that we use to implement cost stacks are a  */
/* linked list, which is used to represent the index table and a  */
/* cost-centre stack, which is simply a doubly linked list struc- */
/* ture with an Index table, a cost centre name and a previous    */
/* stack pointer. These are defined in C.                         */

typedef struct CostCentreStack COSTCENTRESTACK;
typedef COSTCENTRESTACK *CostStackTable;

typedef struct IndexTableItem INDEXTABLEITEM;
typedef INDEXTABLEITEM *IndexTable;

struct IndexTableItem {
                costCentre CostCentre;
```

```
                CostStackTable NextStack;
                IndexTable NextIndexTableItem;
};

struct CostCentreStack {
                costCentre CostCentre;
                int time_ticks;
{Other profiling information - ProfilingInfo}
                CostStackTable PreviousStack;
                IndexTable CostCentreStackIndexTable;
};

/* ============================================================ */


/* ------------------------ COST STACK FUNCTIONS -------------- */

/* There are a number of functions needed for the manipulation of */
/* cost stacks. The supporting functions; InitialiseCostStackTable*/
/* PrintCostStack, AddToCostStackTable, IsInCostStack,            */
/* PreviousCostCentreStack, and NextCostCentreStack are all used  */
/* to implement the function Push, which is the function used in  */
/* GHC to create cost stacks rather than single cost centres. The */
/* Push function is implmented with a few modifications to improve*/
/* the efficiency of the operation.                               */

CostStackTable Push( CostStackTable, costCentre, int );
CostStackTable InitialiseCostStackTable();
void PrintCostStack( CostStackTable );
void PrintAllStacks( CostStackTable );
CostStackTable AddToCostStackTable( CostStackTable, costCentre,
                                    ProfilingInfo );
int IsInCostStack( CostStackTable, costCentre );
CostStackTable PreviousCostCentreStack( CostStackTable );
CostStackTable NextCostCentreStack( CostStackTable, costCentre );

/* ============================================================ */


/* ------------------------ INDEX TABLE FUNCTIONS ------------- */

/* The index table functions are used by the cost stack support   */
/* functions and also by the Push operation.    They are simply   */
/* functions on linked lists but they make the program simpler to */
/* understand at a higher level.The linked list is created in the */
/* same order as sequences: x4^<x3,x2,x1> = <x4,x3,x2,x1>.        */
```

```
IndexTable InitialiseIndexTable ();
CostStackTable IsInIndexTable( IndexTable, costCentre );
void PrintIndexTable ( IndexTable );
IndexTable AddToIndexTable( IndexTable,costCentre,CostStackTable );
IndexTable NextIndexTableItem ( IndexTable );
CostStackTable FindNextStack( IndexTable, costCentre );

/* ============================================================ */

Specification of COST-STACK FUNCTIONS

/* ============================================================ */

CostStackTable Push( CostStackTable cst, costCentre cc,
                     ProfilingInfo stats ) =
if ((cst!=Null) AND ((cst->costCentre)=cc))
  {- Cost centre at the top of stack -}
  Return *(CostStackTable{cc;cst->ProfilingInfo+stats,
                          cst->CostCentreStackIndexTable})
otherwise
  if (cst!=Null) {- ie. Cost centre stack is not empty -}
     x = IsInIndexTable(cst->CostCentreStackIndexTable,cc);
     if (x != NULL)
 return x {- Reference to cost centre in index table -}
     otherwise
 if (IsInCostStack(cst,cc)!=NULL)
            {- ie. In stack already. Optimisation pos -}
    BackList = InitialiseCostStackTable();
    Previous = cst;
    while (Previous->cc != cc)
BackList = AddToCostStackTable(BackList,
                               Previous->CostCentre,
                               Previous->ProfilingInfo);
Previous = PreviousCostCentreStack(Previous);
    stats' = Previous->ProfilingInfo;
    Previous = PreviousCostCentreStack(Previous);
            {- Remove existing occurance -}
    while (BackList != NULL)
Previous = AddToCostStackTable(Previous,
                                    BackList->CostCentre,
                                    BackList->ProfilingInfo);
BackList = BackList->NextIndexTableItem;
    Return AddToCostStackTable(Previous,cc,stats+stats')
 otherwise
    Return AddToCostStackTable(cst,cc,stats)
                  {- Not in stack so just add -}
  otherwise
```

```
 Return AddToCostStackTable(InitialiseCostStackTable(),
                                    cc,stats)
 {- as Cost stack is empty}


CostStackTable InitialiseCostStackTable() =
Return NULL

void PrintCostStack( CostStackTable cst ) =
if (cst = NULL)
SKIP
otherwise
(print cst->CostCentre);
PrintCostStack(cst->NextStack)

void PrintAllStacks( CostStackTable cst ) =
if (cst = NULL)
SKIP
otherwise
PrintCostStack cst;
PrintAllStacks NextIndexTableItem

CostStackTable AddToCostStackTable( CostStackTable cst,
                                    costCentre cc,
    ProfilingInfo stats ) =
if (cst = NULL)
   Return *(CostCentreStack{cc,stats, NULL, NULL})
else
   if (IsInIndexTable(cst->indextable,cc)!=NULL)
      Return FindNextStack(cst->indextable,cc)
   else
              new = *(CostCentreStack{cc,stats,cst,NULL})
      cst->IndexTable =
                          AddToIndexTable(cst->IndexTable,cc,new)
      Return new

int IsInCostStack( CostStackTablecst, costCentre cc ) =
Return (cc in cst)

CostStackTable PreviousCostCentreStack( CostStackTable cst ) =
Return (cst->PreviousStack)

CostStackTable NextCostCentreStack(CostStackTable cst,costCentre cc)
= Return FindNextStack(cst->CostCentreStackIndexTable,cc)


/* ============================================================ */
```

```
Specification of INDEX TABLE FUNCTIONS

/* ============================================================== */

IndexTable InitialiseIndexTable () = Return NULL

CostStackTable IsInIndexTable( IndexTable it, costCentre cc ) =
if (IndexTableItem{cc;nextStack;_} in it)
Return nextStack
otherwise
Return NULL

void PrintIndexTable ( IndexTable it ) =
map print it

IndexTable AddToIndexTable( IndexTable it , costCentre cc ,
                    CostStackTable cst ) =
Return *(IndexTableItem{cc;cst;it})

IndexTable NextIndexTableItem ( IndexTable it ) =
Return (it->NextIndexTableItem)

CostStackTable FindNextStack( IndexTable it, costCentre cc ) =
Return IsInIndexTable(it,cc)
```

Bibliography

[Ajisaka, 1987] T. Ajisaka, *"Studies on Automatic Program Generation using Functional Model of Specification,"* Ph.D Thesis Department of Information Science, Kyoto University. December 1987

[Appel, Duba, MacQueen, 1988] A. W. Appel, B. F. Duba and D. B. MacQueen, *"Profiling in the presence of optimization and garbage collection,"* Part of the New Jersey SML distribution. November 1988

[Armstrong, Williams and Virding, 1993] J. Armstrong, M. Williams and R. Virding, *"Concurrent Programming in Erlang,"* Prentice Hall, ISBN 13-285792-8, 1993

[Benley, 1982] J. L. Bentley, *"Writing Efficient Programs,"* Prentice-Hall Software Series, 1982

[Bentley, 1986] J. L. Bentley, *"Programming Pearls,"* Addison-Wesley Publishing Company, 1986

[Bentley, 1986] J. L. Bentley, *"Programming Pearls — Profilers,"* Communications of the ACM 30, 587-592, 1986

[Bentley, 1988] J. L. Bentley, *"More Programming Pearls,"* Addison-Wesley Publishing Company, 1988

[Bird and Wadler, 1988] R. S. Bird and P. Wadler, *"Introduction to Functional Programming,"* Series in Computer Science, Prentice Hall International. 1988

[Bjerner and Holmstrom, 1989] B. Bjerner and S. Holmstrom, *"A Compositional Approach to Time Analysis of First Order Lazy Functional Programs,"* In Proceedings of the 1989 Conference on Functional Programming Languages and Computer Architecture, pages 157-165. ACM Press, September 1989

[Boldyreff, Burd and Hather, 1995] C. Boldyreff, E. L. Burd, R. M. Hather, R. E. Mortimer, M. Munro and E. J. Younger, *"The AMES Approach to Applications Understanding: a case study,"* International Conference on Software Maintenance, IEEE, France, October 1995

[Bodhuin, 1995] T. Bodhuin, *"An Interaction Paradigm for Impact Analysis,"* M.Sc. Thesis, University of Durham, 1995

[Bratko, 1990] I. Bratko, *"Prolog: Programming for Artificial Intelligence,"* Addison-Wesley, 1990

[Burstall and Darlington, 1977] R. M. Burstall and J. Darlington, *"A transformation system for developing recursive programs,"* Journal of the ACM 24, 1, pp. 44-67, January 1977

[Burn, 87] G. L. Burn, *"Evaluation Transformers — A model for the parallel evaluation of functional languages,"* In Proceedings of FPCA Conference, pages 446-470, ACM, Springer-Verlag, September 1987

[Clayman, Parrott and Clack, 1991] S. Clayman, D. J. Parrott and C. Clack, *"A Profiling Technique for Lazy, Higher Order Functional Programs,"* Research Note RN/92/24, Department of Computer Science, University College London, November 1991

[Clayman, Clack and Parrott, 1995] S. Clayman, C. Clack and D. J. Parrott, *"Lexical Profiling: Theory and Practice,"* Journal of Functional Programming, Volume 5, Part 2, 1995

[Darlington and Burstall, 1976] J. Darlington and R. M. Burstall, *"A system which automatically improves programs,"* Acta Informatica 6, 1, pp. 41-60, 1976

[Ellis, Garigliano and Morgan, 1993] N. R. Ellis, R. Garigliano and R. G. Morgan, *"A New Transformation into Deterministically Parsable Form for Natural Language Grammars,"* In Proceedings of the 3rd international workshop on Parsing Technologies, Tilburg, Netherlands, 1993

[Foubister and Runciman, 1995] S. P. Foubister and C. Runciman, "Techniques for simplifying the visualisation of graph reduction," Functional Programming, Glasgow 1994, pp. 66 - 77, Springer-Verlag 1995

[Garigliano, Morgan and Smith, 1992] R. Garigliano, R. G. Morgan and M. H. Smith, *"LOLITA: Progress Report 1,"* Artificial Intelligence Systems Research Group, Technical Report 12/92, School of Engineering and Computer Science, University of Durham, UK, 1992

[Garigliano and Jones, 1992] R. Garigliano and C. Jones, *"Dialogue Structure Models: An approach to dialogue analysis and generation by computer,"* Technical Report 1/92, School of Engineering and Computer Science, University of Durham, UK, 1992

[Garigliano, Morgan and Smith, 1993] R. Garigliano, R. G. Morgan and M. H. Smith, *"The LOLITA system as a context scanning tool,"* In Proceedings of *Avignon*, 1993

[Garigliano and Tate, 1995] Journal of Natural Language Engineering, Volume 1, R. Garigliano and J. Tate editors, Cambridge University Press, 1995

[Gill and Wadler, 1995] *"Real World Applications of Functional Programs,"* A. Gill and P. Wadler editors. Available via World Wide Web page http://www.dcs.gla.ac.uk/fp/realworld.html, updated 1995

[Goblirsch, 1993] D. M. Goblirsch, *"Mitre Speech Recognition System,"* The Mitre Corporation, Colshire Drive, McLean, Virginia, USA, 1993

[Graham, Kessler and Kusick, 1982] S. L. Graham, P. B. Kessler and M. K. Kusick, *"gprof: a call graph execution profiler,"* ACM Sigplan Notices, 17(6):120-126, Symposium on Compiler Construction, June 1982

[Hanna, Daeche and Howells, 1992] F. K. Hanna, N. Daeche and W. G. J. Howells, *"Implementation of the Veritas design logic,"* In Proceedings of Theorem provers in circuit design; IFIP trans. A-10, pages 77-94. North Holland, 1992

[Hartel and de Jong, 1994] P. H. Hartel and E. K. de Jong, *"Prototyping a smart card architecture in a lazy functional programming language,"* Department of Computer Systems, University of Amsterdam, Technical Report CS-94-08, May 1994

[Hazan and Morgan, 1992] J. E. Hazan and R. G. Morgan, *"The Location of Errors in Functional Programs,"* Technical Report Number 3/92, Artificial Intelligence Research Group, School of Engineering and Computer Science, University of Durham, 1992

[Hazan and Morgan, 1993] J. E. Hazan and R. G. Morgan, *"The location of errors in functional programs,"* In Lecture Notes in Computer Science 749, pp. 135-152, Springer-Verlag, 1993

[Hazan, Jarvis, Morgan and Garigliano, 1993] J. E. Hazan, S. A. Jarvis, R. G. Morgan and R. Garigliano, *"Understanding LOLITA: Program Comprehension in Functional Languages,"* In Proceedings of IEEE Conference on Program Comprehension, IEEE Computer Society Press, pp. 26-34, June 1993

[Hennessy and Patterson, 1990] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach,"* Morgan Kaufmann Publishers, San Mateo, CA, 1990

[Hinchey and Jarvis, 1994] M. G. Hinchey and S. A. Jarvis, *"Concurrent Systems: Formal Development in CSP,"* The McGraw-Hill International Series in Software Engineering, 1994

[Hudak and Fasel, 1992] P. Hudak, J. Fasel and The Haskell Committee, *"Haskell: Version 1.2,"* SIGPLAN Notices 27(5), May 1992

[Hughes, 1989] J. Hughes, *"Why functional programming matters,"* In The Computer Journal, Volume 32, 1989

[Jarvis, Glaser and van Eekelen, 1995] *"Functional Programming Languages in Education,"* S. A. Jarvis, H. Glaser and M. van Eekelen editors. Available via World Wide Web page http://www.cs.kun.nl/fple/ updated April 1996

[Jarvis and Morgan, 1993] S. A. Jarvis and R. G. Morgan, *"Profiling the Profilers: An Historical Review,"* Artificial Intelligence Research Group, University of Durham, July 1993

[Jarvis and Morgan, 1996a] S. A. Jarvis and R. G. Morgan, *"The Results of: Profiling Large-scale Lazy Functional Programs,"* International Workshop on the Implementation of Functional Languages, Gustav-Stresemann-Institut, Bonn-Bad-Godesberg, Germany, Sept. 16th-18th, 1996

[Jarvis and Morgan, 1996b] S. A. Jarvis and R. G. Morgan, *"Profiling Large-scale Lazy Functional Programs,"* In preparation for the Journal of Functional Programming, 1996

[Jarvis, Poria and Morgan, 1995] S. A. Jarvis, S. Poria and R. G. Morgan, *"Understanding LOLITA: Experiences of Teaching Large Scale Functional Programming,"* Symposium on Functional Programming Languages in Education, Mook, Netherlands, December 1995. Proceedings in Lecture Notes in Computer Science LNCS 1022, pp. 103-120, Springer-Verlag, 1995

[Johnson, 1975] S. C. Johnson, *"Yacc — Yet Another Compiler Compiler,"* Murry Hill, New Jersey, USA, 1975

[Jones, 1995] M. Jones, The Haskell Users Gofer System (HUGs), Department of Computer Science, University of Nottingham, UK. 1995

[Jones and Garigliano, 1993] C. Jones and R. Garigliano, *"Dialogue Analysis and Generation: A Theory for Modelling Natural English Dialogue,"* In Proceedings of EUROSPEECH '93, Volume 2, pp. 951, Berlin, Germany. 1993

[Kinloch and Munro, 1994] D. Kinloch and M. Munro, *"Understanding C Programs Using the Combined C Graph Representation,"* Proceedings of the International Conference on Software Maintenance, Victoria, Canada, September 1994

[Launchbury, 1993] J. Launchbury, *"A natural semantics for lazy evaluation,"* in Proceedings of 20th ACM Symposium on Principles of Programming Languages, Charlotte, ACM, 1993

[Long and Garigliano, 1994] D. Long and R. Garigliano, *"Reasoning by Analogy and Causality: A Model and Application,"* Ellis Horwood, 1994

[Loveman, 1977] D. B. Loveman, *"Program improvement by source-to-source transformation,"* Journal of the ACM 24, 1, January 1974, pp. 121-145

[Major, Lapalme and Cedergren, 1991] F. Major, G. Lapalme and R. Cedergren, *"Domain Generating Functions for Solving Constraint Satisfaction Problems,"* Journal of Functional Programming, Vol 1, No. 2, 1991, pp 213-237

[McColl, 1995] W. F. McColl, "The BSP Approach to Architecture Independent Parallel Programming," Programming Research Group, Oxford University, March 1995

[Morgan and Garigliano et al, 1995] R. G. Morgan, R. Garigliano, P. Callaghan, S. Poria, M. Smith, A. Urbanowicz, R. Collingham, M. Constantino, C. Cooper and the LOLITA Group, *"University of Durham: Description of the LOLITA system used in MUC-6,"* Proceedings of the Message Understanding Conference (MUC-6), 1995

[Morgan, Garigliano, Jarvis and Parker, 1994] R. G. Morgan, R. Garigliano, S. A. Jarvis and B. S. Parker, *"Maintenance and Development of Large Scale Lazy Functional Programs,"* Dagstuhl Workshop on Functional Programming in the Real World, organisors R. Giegerich and J. Hughes. Dagstuhl, Germany. May 16-20, 1994

[Morgan, Garigliano, Jarvis and Parker, 1996] R. G. Morgan, R. Garigliano, S. A. Jarvis and B. S. Parker, *"LOLITA: A Large Scale Natural Language Processing System written in Haskell,"* In preparation for the Journal of Functional Programming, 1996

[Morgan and Jarvis, 1995] R. G. Morgan and S. A. Jarvis, *"Profiling Large-scale Lazy Functional Programs,"* In Proceedings of High Performance Functional Computing, A. P. W. Bohm and J. T. Feo Editors, Lawrence Livermore National Laboratory, USA, pp. 222-234, April 1995

[Milner, 1990] R. Milner, M. Tofte and R. Harper, *"The Definition of Standard ML,"* MIT, 1990. ISBN 0-262-63132-6

[Nettleton and Garigliano, 1994] D. J. Nettleton and R. Garigliano, *"Evolutionary Algorithms for Dialogue Optimisation in the LOLITA Natural Language Processor,"* Seminar on Adaptive Computing and Information Processing, University of Durham, UK. January 1994

[Parrott and Clayman, 1990] D. J. Parrott and S. Clayman, *"Report on 'Cost' and 'Debug' primitive extensions to FLIC,"* Department of Computer Science, University College London, 1990

[Partain, 1992] W. Partain, *"The* `nofib` *Benchmark Suite of Haskell Programs,"* Department of Computer Science, University of Glasgow, 1992

[Paulson and Nipkow, 1986] L. C. Paulson and T. Nipkow, University of Cambridge and Technical University, Munich, Isabelle-94 available from FTP sites at both of these sources. Also see "A Gentle Introduction to Isabelle", 1986-

[Peyton Jones, 1987] S. L. Peyton Jones, *"The implementation of functional programming languages,"* Prentice-Hall, 1987

[Peyton Jones, 1992] S. L. Peyton Jones, *"Implementing lazy functional programs on stock hardware: The Spineless Tagless G-machine,"* Journal of Functional Programming 2, 127-202, 1992

[Peyton Jones, Hall and Hammond, 1993] S. L. Peyton Jones, C. V. Hall, K. Hammond, W. D. Partain and P. L. Wadler, *"The Glasgow Haskell Compiler: a technical overview,"* in Joint Framework for Information Technology Technical Conference, Keele 1993

[Peyton Jones and Joy, 1989] S. L. Peyton Jones and M. S. Joy, *"a Functional Language Intermediate Code,"* Internal note 2048, University College London, Department of Computer Science, August 1989

[Peyton Jones and Santos, 1994] S. L. Peyton Jones and A. Santos, *"Compilation by Transformation in the Glasgow Haskell Compiler,"* Department of Computer Science, University of Glasgow, 1994

[Platter and Nievergelt, 1981] B. Platter and J. Nievergelt, *"Monitoring program execution: A survey,"* IEEE Computer Magazine 14, 11, pp.76-93, November 1981

[Runciman and Röjemo, 1996] C. Runciman and N. Röjemo, *"Heap profiling for space efficiency,"* In J. Launchbury and E. Meijer and T. Sheard, editors, 2nd International School on Advanced Functional Programming, Springer LNCS Vol. 1129, August 1996

[Runciman and Wakeling, 1990] C. Runciman and D. Wakeling, *"Problems and proposals for time and space profiling of functional programs,"* In S.L.Peyton Jones, G.Hutton and C.K.Holst, editors, Functional Programming Workshop, Glasgow 1990, pages 237-245. Springer-Verlag: Workshops in Computing, August 1990

[Runciman and Wakeling, 1992a] C. Runciman and D. Wakeling, *"Heap Profiling for Lazy Functional Languages,"* Technical report no. 172, Department of Computer Science, University of York, April 1992

[Runciman and Wakeling, 1992b] C. Runciman and D. Wakeling, *"Heap Profiling of a Lazy Functional Compiler,"* in Functional Programming, Glasgow 1992, J. Launchbury and P. Sansom eds., Springer-Verlag, Workshops in Computing, Ayr, Scotland, 1992

[Runciman and Wakeling, 1993] C. Runciman and D. Wakeling, *"Heap Profiling of Lazy Functional Programs,"* Journal of Functional Programming, Volume 3, Part 2, 1993

[Runciman and Wakeling, 1995] C. Runciman and D. Wakeling (Eds.), *"Applications of functional programming,"* UCL Press, 1995

[Sands, 1991] D. Sands, *"Time analysis, Cost Equivalence and Program Refinement,"* In Proceedings of the Conference on the Foundations of Software Technology and Theoretical Computer Science, pages 25-39. Springer-Verlag, LNCS 560, December 1991

[Sansom, 1994] P.M Sansom, *"Execution Profiling for Non-strict Functional Languages,"* Ph.D Thesis, University of Glasgow, UK, Research Report FP-1994-09, 1994

[Sansom and Peyton Jones, 1992] P. M. Sansom and S. L. Peyton Jones, *"Profiling Lazy Functional Programs,"* In Functional Programming, Glasgow 1992, J. Launchbury and P. M. Sansom, Editors, Springer-Verlag Workshops in Computing, Ayr, Scotland, July 1992.

[Sansom and Peyton Jones, 1994] P. M Sansom and S. L. Peyton Jones, *"Time and space profiling for non-strict, higher-order functional languages,"* Research Report FP-1994-10, University of Glasgow, UK, Nov 1994

[Sansom and Peyton Jones, 1995] P.M Sansom and S.L. Peyton Jones, *"Time and space profiling for non-strict, higher-order functional languages,"* 22nd ACM Symposium on Principles of Programming Languages, San Francisco, California, January 1995

[Selinker, 1969] L. Selinker, *"Language Transfer,"* General Linguistics 9, pp. 69-92, 1969

[Shastri, 1988] L. Shastri, *"Semantic Networks: An evidential Formalisation and it Connectionalist Realisation,"* Morgan Kaufmann, 1988

[Smith, 1995] M. Smith, *"Natural Language Generation in the LOLITA System: An Engineering Approach,"* Ph.D. Thesis, Department of Computer Science, University of Durham, 1995

[Soden and Bock, 1995] A. C. Soden and H. Bock, *"Extracting Characteristics from Functional Programs for Mapping to Massively Parallel Machines,"* Proceedings of High Performance Functional Computing, A.P.W. Boem and J.T. Feo eds., Lawrence Livermore National Laboratory, USA, April 1995

[Sommerville, 1992] I. Sommerville, *"Software Engineering,"* Addison-Wesley Publishing Company, 1992

[Spivey, 1989] J. M. Spivey, *"The Z Notation: A Reference Manual,"* Prentice Hall International Series in Computer Science, 1989

[Thiemann, 1994] P. Thiemann, *"Ebnf2ps,"* Universität Tübingen, Sand 13, D-72076, Germany, 1994

[Tomita, 1986] M. Tomita, *"Efficient Parsing of NL: A Fast Algorithm for Practical Systems,"* KAP, Boston, Ma. USA, 1986

[Turner, 1982] D. A. Turner, *"Recursion equations as a functional language,"* In Darlington et al., editor Functional Programming and its Applications, pages 1-28, Cambridge University Press. 1982

[Turner, 1985] D. Turner, Miranda, FPCA '85, 1985

[UNIX, 1979] UNIX Programmers Manual, Bell Laboratories, Murray Hill, N.J, USA, 1979

[Wadler, 1988] P. Wadler, *"Strictness Aids Time Analysis,"* In Fifteenth Annual ACM Symposium on the Principles of Programming Languages, pages 119-131, January 1988

[Wang and Garigliano, 1992] Y. Wang and R. Garigliano, *"An Intelligent Tutoring System for Handling Errors Caused by Transfer,"* in Lecture notes in Computer Science 608: Proceedings of the 2nd International Conference on Intelligent Tutoring Systems, pp. 395-404, Springer-Verlag, Montreal, Canada. 1992

[Wilf, 1986] H. S. Wilf, *"Algorithms and Complexity,"* Prentice-Hall, 1986

[Zorn and Hilfinger, 1988] B. Zorn and P. Hilfinger, *"A memory allocation profiler for C and LISP programs,"* USENIX 88, pages 223-237, 1988